THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse III - Paul Sabatier
Discipline ou spécialité : Informatique

Présentée et soutenue par Cédric Landet
Le 16 décembre 2009

Titre : Modélisation d'un processeur à exécution simultanée de flots pour le temps réel strict

---

JURY
Professeur Jean Paul Bodeveix
Professeur Bernard Goossens - Rapporteur
Professeur Pascal Sainrat – Directeur de Thèse
Professeur Smail Niar - Rapporteur

Ecole doctorale : EDMITT
Unité de recherche : Institut de recherche en Informatique de Toulouse
Directeur(s) de Thèse : Professeur Pascal Sainrat
Rapporteurs : Professeur GOOSSENS Bernard – Professeur NIAR Smail

Remerciements

Je remercie Christine Rochange pour ces années d'encadrement, pour sa patience et sa disponibilité. Pascal Sainrat pour son accueil dans l'équipe et ses conseils.

Je veux remercier aussi l'équipe du professeur Ungerer qui m'a accueilli lors de mon séjour à Augsbourg. Elle m'a permis de mieux connaître CarCore qui est le centre de mon travail.

Je remercie également Bernard Goosens et Smail Niar d'avoir accepté d'être les rapporteurs de mon travail. Merci pour leurs critiques constructives.

Je remercie encore Jean Paul Bodeveix d'avoir accepté de participer au jury de ma thèse.

Je remercie aussi Tahiry Ratsiambaotra pour son aide précieuse dans l'utilisation de GLISS. Merci pour les modifications qu'il a faites pour adapter cet outil aux besoins de mon étude.

Merci aussi à Hugues Cassé qui a répondu sans se lasser à mes interrogations et résolu mes quelques problèmes avec OTAWA.

Je remercie aussi l'équipe technique de l'IRIT, en particulier François Thiebolt pour sa disponibilité et sa réactivité.

Je veux aussi remercier tous ceux qui m'ont soutenu pendant ces quatres années, mes parents, mes amis, en particulier Paul Lacoste que j'ai sans doute parfois saoulé, Mais aussi Clément Ballabriga pour son écoute quotidienne et sa bonne humeur.

Je veux aussi rendre hommage à Daniel Litaize, professeur à l'université Paul Sabatier, décédé pendant ma thèse et qui m'a transmis, pendant mes années d'études dans cette université, ce goût pour l'architecture des ordinateurs.

Je dédie cette thèse à mon grand père qui a si bien sû encadrer le début de mes études et sans lequel je ne serais pas là aujourd'hui.

Table des matières

1 Introduction **3**

1.1 Besoin de performances des applications temps réel 3
1.2 Les architectures performantes exploitent le parallélisme 4
1.3 Les applications temps réel ont aussi besoin de prévisibilité 6
1.4 Organisation du rapport 6
1.5 Apports et contributions 7

2 Les mécanismes de la performance et leur prévisibilité temporelle **9**

2.1 Le principe de calcul du pire temps d'exécution 9
2.1.1 Les méthodes dynamiques 10
2.1.2 Les méthodes statiques 11
2.1.3 Les méthodes hybrides 20
2.2 Le parallélisme d'instructions 21
2.2.1 Exploiter le parallélisme d'instructions 22
2.2.2 Parallélisme d'instructions et calcul de pire temps d'exécution . . 24
2.3 La hiérarchie mémoire 28
2.3.1 Le principe des mémoires caches 29
2.3.2 La prise en compte de la hiérarchie mémoire dans le calcul du pire temps d'exécution 30
2.4 Le parallélisme de tâches 31
2.4.1 Les processeurs multiflots simultanés 33
2.4.2 Les processeurs multicœurs 39
2.5 Conclusion 43

3 Analyse de CarCore **45**

3.1 Présentation de CarCore 45
3.1.1 Le pipeline 46

3.2 Modélisation de CarCore 49
3.2.1 Les graphes d'exécution 49
3.2.2 Modélisation de CarCore par des graphes d'exécution 53
3.3 Implantation de la modélisation 56
3.3.1 Documentation sur CarCore 57
3.3.2 GLISS 57
3.3.3 OTAWA 63
3.4 Conclusion 69

4 Performances pire cas de l'architecture CarCore 71
4.1 Méthodologie 71
4.1.1 Choix des benchmarks 71
4.1.2 Préparation des benchmarks 73
4.1.3 Analyse des benchmarks utilisés 76
4.2 Résultats 77
4.2.1 Évaluation du modèle de CarCore 77
4.2.2 Impact de quelques paramètres du processeur sur le pire temps d'exécution 81
4.3 Conclusion 90

5 Conclusion et perspectives 91
5.1 Résumé du travail accompli 91
5.2 Perspectives 93
5.2.1 Modifications de CarCore pour mieux exploiter le parallélisme d'instructions 93
5.2.2 Vers une meilleure exploitation du parallélisme de tâche 93

Bibliographie 95

Annexe 107

A Preuves des équations 107
A.1 preuve de l'équation 3.2 107

Résumé

Dans un système temps réel, les tâches doivent se terminer avant une date échéance. Pour les ordonnancer, il est nécessaire de connaître leur pire temps d'exécution. Ces systèmes gagnant en complexité, ils demandent une puissance de calcul de plus en plus grande. Pour faire face à cette demande, on peut utiliser des processeurs qui exploitent, en plus du parallélisme d'instructions, le parallélisme de tâches. C'est-à-dire qu'ils sont capables d'exécuter plusieurs tâches en parallèle. Mais la complexité de ces processeurs nuit à la prévisibilité du pire temps d'exécution des tâches.

CarCore est un processeur conçu par l'équipe du professeur Ungerer de l'Université d'Augsbourg (Allemagne). Il permet l'exécution simultanée de plusieurs tâches au sein d'un même coeur. Il a été conçu pour isoler temporellement une tâche de l'influence des autres tâches qu'il exécute.

Nous proposons une modélisation de ce processeur qui permet l'évaluation du pire temps d'exécution de la tâche temps réel avec des méthodes statiques. Nous mettons en évidence les deux sources de surestimation liées à notre modèle qui peuvent entraîner ponctuellement des surestimations de respectivement 1 et 3 cycles. En analysant ces sources de surestimation, nous montrons que des méthodes d'analyse statique ne semblent pas être suffisantes pour les supprimer.

Nous proposons aussi une analyse de l'impact de quelques modifications du processeur sur le pire temps d'exécution estimé. Ces paramètres sont en particulier la taille de la fenêtre d'instructions et la longueur du pipeline. Pour cette dernière, nous envisageons l'ajout d'étages en 4 endroits significatifs du pipeline.

Notre travail ouvre sur des perspectives comme des propositions de modification du pipeline qui permettront l'exécution de plusieurs tâches temps réel ou encore l'augmentation des performances du processeur sans que la précision de l'évaluation du pire temps d'exécution n'en souffre.

Abstract

In a real-time system, tasks must be completed before a deadline date. For the schedule, it is necessary to know their worst case execution time. These systems become more complex, they require an increasing power. To meet this demand, we can

use processors that operate in addition to instruction-level parallelism, task-level parallelism. Thus, they are capable of performing several tasks in parallel. But the complexity of these processors impact the tightness of the worst case execution time.

CarCore is a processor designed by the team of Professor Ungerer, University of Augsburg (Germany). It allows the simultaneous execution of multiple tasks within a single core. It was designed to temporally isolate a task of the influence of other tasks it performs.

We propose a model of this processor that allows the evaluation of worst case execution time of real-time tasks with static methods. We highlight the two sources of overstatement related to our model which can occasionally cause overestimation of respectively one and three cycles. In analyzing these sources of overestimation, we show that static analysis methods do not seem to be sufficient to remove them.

We also offer an analysis of the impact of some modifications of the processor core on the estimated worst case execution time. These parameters are, in particular, the size of the instructions window and the length of the pipeline. For the latter, we are considering the addition of stages in 4 significant areas of the pipeline.

Our work opens perspectives such as amendment of the pipeline that will allow the execution of several real-time tasks or increasing processor performance without decreasing the precision of the evaluation of worst case execution time.

Chapitre 1

Introduction

Certains systèmes informatiques doivent, en plus de fournir un résultat juste, le donner avant une date échéance appelée contrainte temporelle [113]. Ce sont les systèmes temps réel. Ils sont de plus en plus présents, dans l'industrie comme sur les chaînes de production, dans les transports pour piloter les systèmes de freinage de nos voitures ou encore dans l'avionique. Ils sont aussi présents dans nos loisirs comme pour la lecture d'une vidéo. Dans ce dernier cas, par exemple, chaque image doit être lue, décodée et affichée en un vingt cinquième de seconde (pour un DVD) pour que le débit vidéo soit assuré.

On distingue deux grandes classes de systèmes temps réels :

- les systèmes temps réel **souples** pour lesquels le non respect de certaines échéances n'entraîne pas de dysfonctionnements importants du système même si la qualité du service fourni est dégradée. Parmi ceux-ci, on peut citer les systèmes de diffusion de vidéo pour lesquels le non affichage d'une image dégrade la qualité de la réception mais ne remet pas en cause la visualisation elle-même.
- les systèmes temps réel **stricts** dans lesquels aucune violation des contraintes temporelles ne peut être tolérée. Parmi ceux-ci, certains sont **critiques**, c'est-à-dire que leur dysfonctionnement peut entraîner des pertes humaines ou des catastrophes.

1.1 Besoin de performances des applications temps réel

Les systèmes temps réel offrent de plus en plus de fonctionnalités et les traitements qu'ils nécessitent augmentent en complexité. La demande de puissance de calcul qui en découle s'accroît d'autant.

Le décodage vidéo fournit un bon exemple : l'augmentation de la résolution des films lors du passage du support DVD[1], 720*576 (PAL[2]), au support Blu-Ray, 1920*1080 (1080p), engendre une augmentation du volume de données à traiter d'un facteur 5 pour chaque image.

A cela, il faut ajouter un surcoût lié à l'encodage qui est plus complexe sur les Blu-Ray (MPEG2[3], H264/MPEG4-AVC[4] et SMPTE[5]-VC1) que sur les DVD (MPEG-2). De plus, les mécanismes de protection anti-copie (DRM[6]), Macrovision et CSS[7] pour les DVD, contre AACS[8] pour le Blu-Ray, gagnent aussi en complexité à mesure qu'ils sont déjoués.

Tous ces traitements, doivent être réalisés dans un temps voisin pour les DVD ou les Blu-Ray ; le DVD affichant 25 images par seconde et le Blu-Ray en affichant, dans le même temps, 25 voire 30 selon le standard.

La quantité de données à traiter est donc augmentée et ce traitement est plus complexe alors que le temps pour réaliser ce traitement est, au mieux, inchangé. Ces augmentations du volume de données à traiter et de la complexité des traitements ne peuvent pas être absorbées sans une révision de la puissance de calcul du processeur qui fait le traitement.

1.2 Les architectures performantes exploitent le parallélisme

Dans la conception des processeurs modernes, on cherche à les rendre de plus en plus performants [55, 56, 57, 58, 112], c'est-à-dire à leur faire exécuter le plus possible d'instructions par unité de temps. Pour cela, on peut agir principalement sur deux aspects :

- augmenter la fréquence,
- faire exécuter au processeur plusieurs instructions en même temps : c'est le parallélisme.

[1] Digital Video Disc
[2] Phase Alternate Line
[3] Moving Picture Experts Group
[4] Advanced Video Coding
[5] Society of Motion Picture and Television Engineers
[6] Digital Rights Management
[7] Content Scrambling System
[8] Advanced Access Content System

L'augmentation de la fréquence est, aujourd'hui, devenue difficile. Le parallélisme est un facteur intrinsèque des applications. On le trouve à plusieurs niveaux :

- au niveau de la conception de l'application, lorsque le problème peut être structuré en plusieurs tâches communicantes qui collaborent pour mettre en œuvre un traitement. Chaque tâche donne lieu à un flot de contrôle et ceux-ci peuvent être exécutés indépendamment aux synchronisations près. C'est le **parallélisme de tâches** (TLP[9]).
- au niveau des données : une application nécessite parfois qu'un traitement identique soit appliqué sur plusieurs données. C'est le cas par exemple de nombreux traitements sur des matrices comme l'augmentation de la luminosité dans une image bitmap. C'est le **parallélisme de données**.
- dans le code même du programme, au sein du même flot de contrôle, les instructions qui sont indépendantes entre elles peuvent être exécutées simultanément par les processeurs qui le permettent. C'est le **parallélisme d'instructions** (ILP[10]).

Pour accroître leur puissance de calcul, les processeurs modernes cherchent à exploiter tant le parallélisme d'instructions, que le parallélisme de tâches. Le parallélisme de données n'apparaît que pour certaines applications comme le traitement d'images et des processeurs spécifiques ont été spécialement conçus pour ce type de traitements. Dans la plupart des processeurs génériques, des unités fonctionnelles ont aussi été ajoutées pour traiter les jeux d'instructions MMX[11] ou SSE[12] par exemple, qui permettent d'exploiter du parallélisme de données.

Pour mieux examiner les mécanismes architecturaux qui permettent d'exploiter ces types de parallélisme, on peut catégoriser les ressources internes d'un processeur selon un critère d'activité. Nous distinguerons :

- les **ressources de stockage** qui contiennent des données mais ne les modifient pas. Ce sont les registres, les tampons, les files et les tables…
- des **ressources d'action** ou **de traitement**, comme les unités fonctionnelles par exemple.

Nous utiliserons cette distinction dans la suite de ce rapport pour expliquer comment ces structures permettent d'exploiter le parallélisme mais aussi pour exprimer les problèmes de modélisation.

[9] Task Level Parallelism
[10] Instruction Level Parallelism
[11] officieusement, ces initiales pourraient signifier : MultiMedia eXtension, Multiple Math eXtension ou encore Matrix Math eXtension
[12] Streaming SIMD Extensions

1.3 Les applications temps réel ont aussi besoin de prévisibilité

Le respect des contraintes temporelles doit être garanti a priori pour les systèmes temps réel stricts. Pour ce faire, l'ordonnancement des tâches, c'est à dire le choix de l'ordre dans lequel elles seront exécutées, est généralement calculé hors ligne, lors de la conception du système. Ce calcul nécessite de connaître le **pire temps d'exécution** (WCET[13]) de chaque tâche à ordonnancer.

La possibilité d'évaluer, a priori, une borne supérieure du pire temps d'exécution des tâches d'un système est appelé **prévisibilité**. Les systèmes temps réel doivent être totalement prévisibles. L'évaluation du pire temps d'exécution n'est pas triviale. Ce temps dépend à la fois du logiciel et de l'architecture matérielle qui exécute la tâche.

Pour être utilisable dans le cadre du temps réel strict, les pires temps d'exécution évalués doivent toujours être des *bornes supérieures* du pire temps d'exécution réel. Leur calcul ne peut se faire qu'en utilisant des méthodes statiques. Or ces méthodes nécessitent une modélisation de l'architecture matérielle qui va exécuter les tâches. La précision de ce modèle conditionne la précision de l'évaluation. Or, les mécanismes architecturaux qui permettent d'exploiter le parallélisme augmentent la complexité de cette modélisation et par là même, la difficulté à évaluer des pire temps d'exécution de manière précise. En un mot, la complexité du processeur, si elle accroît ses performances, nuit à la prévisibilité.

1.4 Organisation du rapport

Dans ce rapport, nous allons analyser le comportement d'un processeur à flots multiples simultanés destiné à exécuter une tâche temps réel strict en concurrence avec d'autres tâches qui n'ont pas de contraintes de temps. Ce processeur s'appelle CarCore et a été conçu par l'équipe du professeur Ungerer de l'Université d'Augsbourg (Allemagne). Puis nous étendons notre étude à une analyse de l'impact de certains composant architecturaux, comme la taille de la fenêtre d'instructions ou du pipeline, sur le pire temps d'exécution évalué.

La suite de ce rapport est organisée de la manière suivante. Le chapitre 2 présente les mécanismes de la performance ainsi que leur prévisibilité temporelle. Il détaille le principe de calcul du pire temps d'exécution et montre comment sont pris en compte

[13]Worst Case Execution Time

les mécanismes qui permettent d'exploiter le parallélisme d'instructions. Il introduit la hiérarchie mémoire ainsi que le parallélisme de tâches.

Le chapitre 3 présente le processeur CarCore et montre comment nous l'avons modélisé grâce à une méthode basée sur la construction de graphes d'exécution. Notre implantation du modèle est aussi détaillée dans cette section.

L'évaluation de notre modèle de CarCore ainsi que celle l'évaluation de l'impact des paramètres architecturaux sur le pire temps d'exécution évalué sont faites dans le chapitre 4.

Ce rapport est conclu par le chapitre 5 qui résume notre travail et présente les perspectives pour de futurs travaux.

1.5 Apports et contributions

Notre travail apporte une modélisation par des graphes d'exécution de la taille variable des instructions, du double pipeline, de la politique d'ordonnancement (à priorité fixe), des accès mémoires faits en deux phases et des instructions microcodées. Les résultats expérimentaux ne montrent jamais de surestimation du pire temps d'exécution des programmes analysés en utilisant ce modèle. Ceci est une condition nécessaire pour son utilisation dans le cadre du temps réel strict. D'autre part, les surestimations mises en évidence sont limitées, ce qui est important pour le dimensionnement des systèmes.

Notre analyse de l'impact de certains éléments architecturaux sur le pire temps d'exécution apporte des éléments utiles pour de futurs travaux, en particulier en vue d'une meilleure exploitation du parallélisme dans des processeurs du type de CarCore.

-

Chapitre 2

Les mécanismes de la performance et leur prévisibilité temporelle

2.1 Le principe de calcul du pire temps d'exécution

Le temps d'exécution d'une tâche dépend principalement de deux facteurs : le *jeu de données en entrée* de la tâche, car celui-ci détermine le chemin de contrôle suivi dans le programme lors de son exécution, et l'*architecture matérielle* qui exécute le programme.

Le WCET[14] est le pire des temps d'exécution du programme quel que soit le jeu de données en entrée. Il ne dépend que de l'architecture matérielle qui exécute le programme, même si, comme nous le verrons, l'exploitation du parallélisme de tâches contrarie cette affirmation.

Le WCET estimé doit *toujours être une borne supérieure* du WCET réel, toute sous-estimation pouvant entraîner le non respect d'une contrainte temporelle. Cependant, pour que le pire temps estimé soit utilisable, la surestimation doit rester la plus faible possible afin de ne pas conduire à un surdimensionnement inutile des systèmes.

La figure 2.1 représente les temps d'exécution possibles pour une tâche donnée (les valeurs possibles sont dans la plage grisée). La plus petite valeur, notée BCET[15], est

[14]Worst Case Execution Time
[15]Best Case Execution Time

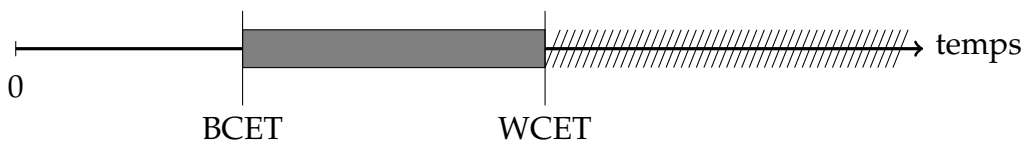


FIG. 2.1 – Le temps d'exécution d'une tâche

le meilleur temps d'exécution possible. La plus grande valeur, notée WCET, est le pire temps d'exécution réel. Toute estimation du WCET doit être supérieure à cette dernière et donc se situer dans le zone hachurée.

Il existe trois classes de méthodes pour évaluer le WCET d'un programme : les méthodes dynamiques, les méthodes statiques et les méthodes hybrides.

2.1.1 Les méthodes dynamiques

Ces méthodes consistent à exécuter le programme sur le processeur cible pour connaître son temps d'exécution. La mesure peut se faire par des équipements externes spécifiques comme un analyseur logique ou en utilisant des compteurs internes présents dans certains processeurs comme le Pentium d'Intel. On peut aussi utiliser un simulateur qui modélise le processeur cible. La simulation se fait cycle par cycle. Une difficulté consiste alors dans la validation du simulateur qui doit être précis et fiable [30], ce qui peut entraîner un surcoût de développement. De plus, il est parfois impossible de connaître exactement la micro-architecture du proceseur.

Pour être sûres, les mesures nécessitent de connaître le jeu de données qui produit le WCET ou, à défaut, de faire des mesures pour tous les jeux de données possibles. Ces jeux de données sont souvent très difficiles à déterminer et faire des mesures pour tous les jeux de données est long (et demande beaucoup de puissance de calcul). On peut aussi laisser à l'utilisateur le soin de fournir ce jeu de données pour son programme. Cette solution se limite aux applications simples et au matériel simple. Des méthodes de génération automatique des jeux de données par analyse des programmes ont été proposés, basées sur des algorithmes évolutionnistes [87] et sur l'algorithme du recuit simulé [117]. D'autres outils tentent de couvrir tous les chemins possibles [65, 126]. D'autres algorithmes génétiques [84, 113] sont utilisés. Toutes ces méthodes ne garantissent pas l'obtention du jeu de données qui mène au pire temps d'exécution. Pour compenser ce manque de certitude, des travaux ont cherché à qualifier la confiance que l'on peut avoir dans les résultats obtenus [44, 46, 62]. Après une analyse de la couverture des chemins suivis, Groß [45] tente de prédire la validité du pire temps d'exécution calculé. Il utilise un algorithme évolutionniste pour générer les jeux de test. L'auteur utilise des méthodes mathématiques complexes [39] pour prédire la proportion des chemins de contrôle couverts par les jeux de tests. Les critères retenus pour cette prédiction sont déterminés de manière empirique et varient selon le système.

Le recours à des jeux de test symboliques [10, 79] permet de couvrir tous les chemins de contrôle du programme. L'idée de base est de partir d'un jeu de données in-

connu et de le propager dans le chemin de contrôle en explorant toutes les branches. Pour cela, le jeu d'instructions du processeur est étendu afin qu'il puisse réaliser des opérations avec des opérandes inconnus. Par exemple, la somme entre un opérande de valeur connue et un autre de valeur inconnue donne un résultat inconnu. Lorsqu'un branchement est rencontré, s'il est conditionné à un opérande inconnu, les deux branches sont explorées. Cette solution implique l'utilisation d'un simulateur logiciel et est coûteuse en calcul. Une solution permettant de réaliser une exécution symbolique est présentée dans [78]. Pour réduire l'espace à explorer dans le flot de contrôle, des annotations [85] de l'utilisateur peuvent être utilisées comme les bornes de boucles, les valeurs attendues pour les paramètres de sous-programmes ou des pronostics sur les tests de branchements conditionnels.

2.1.2 Les méthodes statiques

Les méthodes statiques se basent sur une analyse du code binaire ou du code source du programme. Cette analyse se fait sur des représentations du programme. Elle a généralement lieu en deux temps : une **analyse de flots** et une **analyse temporelle**, en plus du calcul proprement dit.

Représentation du flot de contrôle

La brique de base de la représentation de programmes est le **bloc de base**. C'est une séquence d'instructions qui n'a qu'un seul point d'entrée et un seul point de sortie. La figure 2.2 présente le code d'une fonction. La colonne de gauche contient le code en langage C. La colonne de droite montre une traduction possible de ce code en assembleur ARM. On dénombre donc sept blocs de base, le dernier étant un bloc vide qui constitue la sortie de la boucle. Nous numéroterons ces blocs de base de BB_0 à BB_6. Sur la figure, l'étiquette portant le nom bu bloc est placée à hauteur de la première ligne du bloc.

Deux types de représentation des programmes sont utilisées dans le cadre du calcul de pire temps d'exécution : les **arbres syntaxiques** et **les graphes de flots de contrôle**.

Les arbres syntaxiques sont construits à partir du code source du programme écrit dans un langage de haut niveau. Ce sont des arbres dont les nœuds représentent les structures algorithmiques du programme et dont les feuilles correspondent aux blocs de base. La figure 2.3(a) montre l'arbre syntaxique qu'on peut construire à partir du code en langage C de la figure 2.2. On distingue trois types de nœuds et deux types de feuilles :

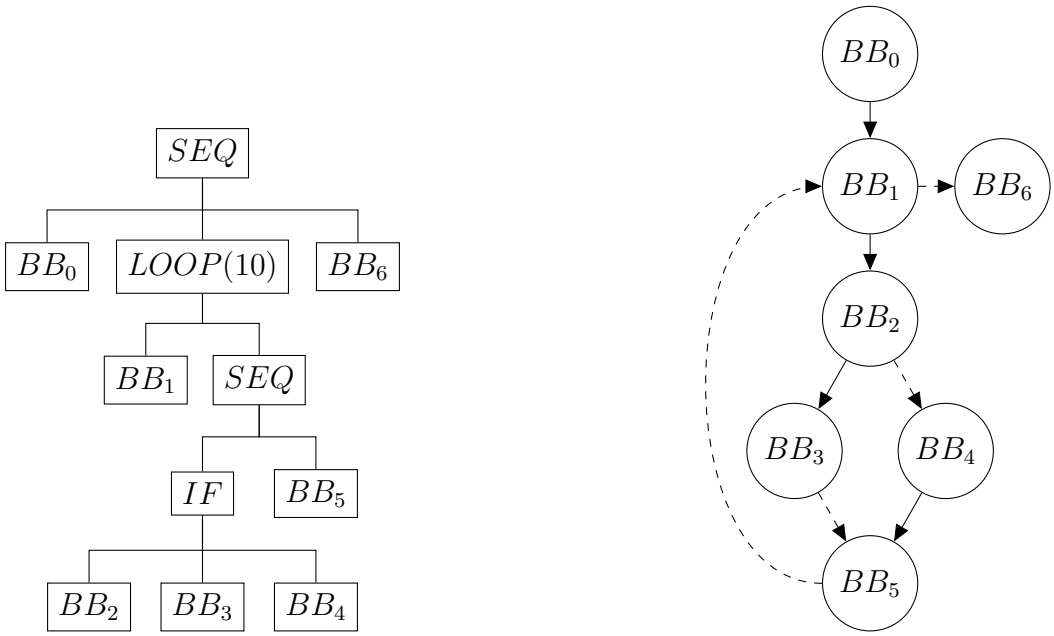
```
#define TAILLE 10

int somme(int *tab){
  int i,sum;

  sum=0;
  i=0;
  while (i<TAILLE)
  {
    if (tab[i]<0)
    {
      sum -= tab[i];
    } else {
      sum += tab[i];
    }
    i++;
  }
  return sum;
}
```

```
somme:              BB0
 mov r0,#0 ;sum
 adr r1,tab
 mov r2 #0 ;i

boucle:             BB1
 cmp r2,#40
 bge fin

 ldr r3,[r1,r2]     BB2
 cmp r3,#0
 bge else

 sub r0,r0,r3       BB3
 b  suite

else:               BB4
 add r0,r0,r3

suite:              BB5
 add r2,r2,#4
 b  boucle

fin:                BB6
```

FIG. 2.2 – Code source d'un programme analysé

- les nœuds SEQ : ils possèdent au moins un fils et représentent la mise en séquence de leurs sous-arbres fils,
- les nœuds $LOOP$: ils représentent les structures itératives, ont deux fils qui sont les sous-arbres *test* et *corps* de la boucle,
- les nœuds IF : ils représentent les structures conditionnelles et ont, comme sous-arbres, les branches *then* et *else* en plus de la branche *test*,
- les feuilles $CALL$: eles représentent les *appels* de sous-programmes,
- les autres feuilles sont les blocs de base.

Les graphes de flots de contrôle sont construits à partir du code de bas niveau du programme (assembleur ou bytecode) [8] en utilisant soit un compilateur modifié [80], soit un outil dédié à la manipulation de code de bas niveau, comme dans SALTO [106] ou OTAWA [20]. Puschner envisage aussi, dans [100], l'obtention du graphe de flot de contrôle à partir du code de haut niveau du programme, mais le compilateur ne doit pas faire d'optimisations. Ces graphes décrivent tous les enchaînements possibles de



(a) Arbre syntaxique (b) Graphe de flot de contrôle

FIG. 2.3 – Représentation du flot de contrôle d'un programme

blocs de base. Leurs nœuds sont les blocs de base et leurs arcs représentent les liens de précédence entre les nœuds. La figure 2.3(b) représente le graphe de flot de contrôle construit à partir du code assembleur de la figure 2.2. On distingue deux types d'arcs : ceux qui représentent des branchements pris sont en pointillés sur la figure, les autres représentent une exécution en séquence.

Les deux types de représentation ont chacun leurs avantages et leurs inconvénients : les arbres syntaxiques contiennent plus d'informations que les graphes de flot de contrôle, mais ils ne tiennent pas compte des éventuelles transformations faites par les compilateurs [124, 123] et excluent l'utilisation de certaines instructions comme les `goto`, les `break` ou les `continue`. De plus, on ne dispose pas toujours du code source des programmes à analyser, en particulier dans l'industrie. En pratique, les deux représentations sont souvent utilisées ensemble [30], ce qui nécessite de s'assurer qu'une correspondance existe bien entre elles, ce qui impose des restrictions sur le code source déjà citées pour les arbres syntaxiques. Engblom et al. ont proposé dans [37] une approche, appelée **co-transformation**, pour établir une correspondance entre les informations provenant du code source de haut niveau et le code objet optimisé. Le principe de base est la spécification dans un langage dédié appelé ODL[16] des optimisations faites par le compilateur. Les informations de haut niveau sont alors transformées en paral-

[16]Optimization Description Language

lèle avec l'optimisation du code, au prix de modifications minimes du compilateur. Une autre méthode, proposée par Kirner et Puschner dans [64], consiste à transformer les informations de flot de contrôle à l'intérieur même du compilateur, par l'introduction d'instructions dans le code intermédiaire RTL[17]. Ces instructions sont transformées en même temps que sont effectuées les optimisations par le compilateur.

Obtention des informations de flot

Les représentations exposées ci-dessus ne suffisent pas pour le calcul sûr et précis du pire temps d'exécution du programme. En particulier, elles n'indiquent pas que les chemins d'exécution sont de taille finie. Elles doivent être complétées par des indications sur le comportement dynamique du programme, en particulier :

- une **borne supérieure du nombre d'itérations des boucles**. Cette information est le plus souvent associée aux boucles sous la forme d'une constante [101, 47]. Ainsi, la constante associée au nœud $LOOP$ dans la figure 2.3(a) représente le plus grand nombre d'itérations de la boucle,
- des contraintes sur les choix d'une branche dans les structures conditionnelles. Elles sont utiles lorsque les choix dépendent d'un paramètre qu'on sait constant,
- des **chemins d'exécution infaisables**, ce qui permet de restreindre l'espace à explorer dans la recherche du chemin le plus long et ainsi d'optimiser le temps de cette recherche. Ce sont des chemins qui apparaissent dans les représentations du programme mais qui ne correspondent pas à une exécution réelle. Par exemple, deux branches peuvent s'exclure mutuellement.

La collecte de ces informations se fait le plus souvent par la fourniture par l'utilisateur d'**annotations** [25, 93, 101] ou bien interactivement [67]. On peut aussi dériver certaines de ces informations automatiquement. Les travaux décrits dans [52] utilisent l'analyse de flot de données pour identifier le nombre d'itérations des boucles ainsi que les chemins infaisables à l'intérieur de celles-ci. Ferdinand et al. [40] utilisent une méthode d'**interprétation abstraite** sur le contenu des registres du processeur pour identifier les chemins d'exécution infaisables. Ermedahl et Gustafsson [47] utilisent une méthode d'interprétation abstraite sur le code source du programme pour obtenir l'ensemble des informations de flot de contrôle. Dans des travaux plus récents [48], ils proposent d'utiliser l'interprétation abstraite pour analyser les mises à jour des valeurs des variables d'indice pour déterminer les bornes de boucles.

[17] Register Transfert Language

L'identification des chemins infaisables ou mutuellement exclusifs peut aussi se faire par **exécution symbolique** comme proposé dans [3] et [77]. Les chemins impossibles sont alors reconnus car ils ne peuvent jamais valider (ou invalider) le test de branchement qui mène à eux. Certains travaux utilisent l'exécution symbolique sur des langages spécifiques [75] ou des langages temps réel [68] pour déterminer ces chemins. Une autre manière de déterminer les chemins infaisables est de construire les chemins faisables [2].

D'autre travaux utilisent des contraintes pour identifier et exprimer un ensemble de chemins infaisables [53]. Dans ces contraintes, les variables sont les blocs de base du graphe de flot de contrôle ainsi que les arcs du graphe de flot de contrôle dont le nœud source se termine par un branchement conditionnel. Les contraintes permettent de spécifier des liens entre ces variables.

Chen et al. proposent de décrire les conflits entre deux chemins sous forme d'un couple (nœud,arc) ou (arc,arc) [26]. Les numéros de nœuds correspondent à une affectation d'une variable et les arcs à des ensembles de valeurs qui ne contiennent pas la valeur de l'affectation ou qui s'excluent mutuellement. Les conditions incompatibles permettent de détecter les conflits. Par exemple, l'affectation $x = 2$ est en conflit avec la branche qui est exécutée quand $x > 3$ et les descriptions des deux branches $x > 3$" et $x < 2$ sont en conflit.

L'**exécution abstraite** [49] peut aussi permettre de détecter les chemins infaisables. Elle se distingue de l'interprétation abstraite par sa façon d'utiliser des ensembles de valeurs pour les variables (au lieu de considérer toutes les valeurs possibles). Cette méthode distingue trois catégories de chemins infaisables :

- les *nœuds infaisables* qui sont des blocs de base qui ne sont jamais exécutés,
- les *paires de nœuds infaisables* qui sont constitués de blocs de base qui s'excluent mutuellement,
- les *chemins infaisables* qui sont constitués de blocs de base qui ne sont jamais exécutés en séquence.

Enfin, oRange [14] est un outil qui permet de déterminer les informations de flots, en particulier les bornes de boucles en combinant l'analyse de flot et l'interprétation abstraite. Il gère les nids de boucles, les appels non récursifs de fonctions, et les conditions de boucles simples, ainsi que les incréments de boucle par addition, soustraction et multiplication. Pour chaque boucle, oRange produit deux résultats :

- `max` est le nombre maximum d'itérations d'une boucle.
- `total` est le nombre total d'exécutions du corps de la boucle, pour l'ensemble des appels de celle-ci.

La détermination de ces valeurs se fait en trois étapes :

- identification et normalisation des boucles, étape au cours de laquelle les variables d'incrément et les expressions de sortie de boucle sont identifiées.
- construction des représentations abstraites de max et total pour chaque boucle.
- calcul des expressions max et total par propagation des expressions construites à l'étape précédente.

Bien qu'oRange ne soit pas capable de gérer certains cas, comme certains appels récursifs de sous-programmes, il donne de bons résultats en termes de précision des bornes calculées. Il ne peut pas produire de sous-estimation.

La détermination automatique des informations de flot, si elle évite les erreurs humaines toujours possibles lors de l'annotation du code par le programmeur, ne permet pas toujours de déterminer toutes les informations nécessaires au calcul du pire temps d'exécution. Les annotations sont donc indispensables.

Les méthodes de calcul

Une fois les informations de flot de contrôle obtenues, il faut trouver le plus long chemin d'exécution à partir de leurs représentations. Il existe plusieurs méthodes pour cela.

La connaissance des pires temps d'exécution des blocs de base permet d'associer ces temps aux nœuds du graphe de flot de contrôle. On obtient un graphe valué avec un seul point d'entrée et un seul point de sortie. On peut alors utiliser l'algorithmique sur les graphes pour rechercher le plus long chemin d'exécution dans le graphe de flot de contrôle [54, 114, 38, 78, 115]. Puis on vérifie que le chemin ainsi déterminé n'est pas un chemin infaisable. S'il l'est, on l'exclut du graphe et on recommence la recherche. On itère ce processus jusqu'à trouver un chemin faisable.

Des méthodes basées sur les arbres syntaxiques ont été proposées [101, 27, 29, 63, 94, 95]. Le principe de base consiste en un parcours de l'arbre qui part des feuilles vers la racine. Le pire temps d'exécution de chaque nœud est ensuite calculé récursivement :

- le pire temps d'exécution d'un nœud SEQ est la somme des pires temps d'exécution de ses sous-arbres,
- le pire temps d'exécution d'un nœud IF est le maximum des temps d'exécution des branches then et else auquel on ajoute le temps d'exécution du test,
- le pire temps d'exécution d'un nœud $LOOP$ est le pire temps d'exécution du corps de la boucle auquel on ajoute le temps du test, le tout pondéré par le nombre d'itérations de la boucle. Il faut encore ajouter le temps d'exécution du

$$
\begin{array}{ll}
\textit{Entres} & \textit{Sorties} \\
n_0 = 1 & n_0 = n_{0,1} \\
n_1 = n_{0,1} + n_{5,1} & n_1 = n_{1,6} + n_{1,2} \\
n_2 = n_{1,2} & n_2 = n_{2,3} + n_{2,4} \\
n_3 = n_{2,3} & n_3 = n_{3,5} \\
n_4 = n_{2,4} & n_4 = n_{4,5} \\
n_5 = n_{3,5} + n_{4,5} & n_5 = n_{5,1} \\
n_6 = n_{1,6} &
\end{array}
$$

FIG. 2.4 – Exemple de système de contraintes généré par la méthode d'énumération des chemins implicites

test qui correspond à la sortie de la boucle.
On obtient ainsi un arbre temporel [100] qui contient les pires temps d'exécution calculés aux différents nœuds de l'arbre.

Une autre méthode est l'énumération de chemins implicites (IPET[18]). Elle est majoritairement utilisée dans les travaux sur l'analyse statique [41, 74, 91, 102]. Elle s'appuie sur la transformation du graphe de flot de contrôle en un système de contraintes. Le jeu de contraintes comporte deux parties : la première partie exprime la structure du graphe, la seconde partie modélise les informations de flots.

La figure 2.4 donne le système de contraintes généré par la méthode d'énumération des chemins implicites pour le graphe de flot de contrôle de la figure 2.3(b). Chaque bloc est exécuté autant de fois que les arêtes qui permettent d'y accéder et autant de fois que celles qui permettent d'en sortir. Le nombre d'exécution du bloc b est noté n_b et le nombre d'exécution d'une arête allant du bloc b au bloc c est noté $n_{b,c}$. A ce système, il faut ajouter l'équation qui modélise le nombre d'itérations de la boucle : $n_2 \leq n_{0,1} * 10$.

Étant donné le système de contraintes, on cherche à maximiser l'expression du pire temps d'exécution :

$$WCET = \sum_i n_i \times \omega_i$$

où les ω_i sont les pires temps d'exécution des blocs de base évalués dans l'analyse temporelle. La maximisation de cette expression fait appel aux méthodes de résolutions de programmes linéaires en nombres entiers (ILP[19]). Le temps d'analyse est fonction de la

[18] Implicit Path Enumeration Technique
[19] Integer Linear Programming

complexité du système [118].

De plus, cette méthode permet d'exprimer l'exclusion mutuelle entre deux chemins : la contrainte $n_\alpha + n_\beta \leq 1$ indique que les nœuds b_α et b_β s'excluent mutuellement [74].

Les méthodes basées sur le code source ne peuvent pas prendre en compte les optimisations faites par les compilateurs, ce qui peut poser des problèmes de précision, voire une sous-estimation du pire temps d'exécution évalué. De plus, le code source des applications n'est pas toujours disponible. Dans la méthode de recherche de chemins dans les graphes, on doit recommencer l'opération tant que le chemin déterminé n'est pas un chemin faisable. La méthode par énumération des chemins implicites ne manipule que les chemins possibles. et travaille sur les blocs de base qui sont déterminés à partir du programme binaire. C'est la méthode la plus utilisée. C'est aussi celle que nous avons choisi d'utiliser dans ce rapport.

L'analyse de bas niveau

Cette étape consiste à calculer les pire temps d'exécution des blocs de base. C'est lors de cette phase de l'analyse que sont pris en compte les éléments architecturaux qui influent sur le temps d'exécution du programme.

A cause de la structure pipelinée des processeurs modernes (voir section 2.2), ce temps dépend du contenu du pipeline au début de l'exécution du bloc. Ce contenu est directement conditionné par les blocs de base qui se sont exécutés avant celui que l'on traite (ces blocs constituent le "préfixe") à cause des dépendances de données et des conflits d'accès aux ressources comme les unités fonctionnelles. Dans les processeurs à ordonnancement dynamique, les blocs qui suivent le bloc considéré (ils constituent le "suffixe") peuvent aussi jouer un rôle.

Dans la figure 2.5, le cas (a) montre un bloc exécuté isolément. Le cas (b) le montre précédé d'un préfixe : on peut voir que son temps d'exécution est modifié. Le cas (c) montre le même bloc avec un préfixe et un suffixe, le temps d'exécution est encore différent. On peut ainsi observer des variations pour chaque couple (préfixe/suffixe). Pour prendre en compte tous les contextes possibles, l'analyse temporelle de chaque bloc de base est faite pour tous ses couples (préfixe/suffixe) possibles.

La variation du temps d'exécution des blocs de base est liée au processeur qui exécute le programme. Ainsi, par exemple, le pipeline d'exécution permet qu'une instruction d'un bloc préfixe influe sur le temps d'exécution du bloc qu'on considère. Ces effets temporels peuvent être dus à des blocs exécutés longtemps avant le bloc courant.

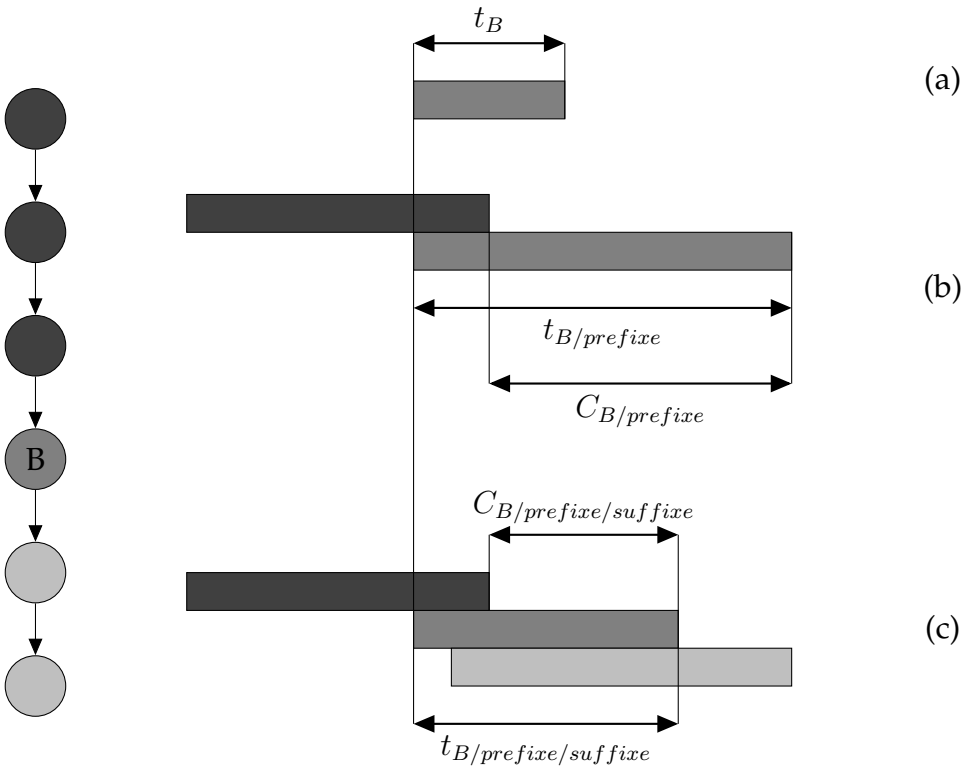


FIG. 2.5 – Préfixe et suffixe d'un bloc de base

Ces "effets longs" ont été mis en évidence par Jacob Engblom dans [36]. Il a montré que la taille du préfixe à considérer n'est pas bornable. De même la présence de mémoire caches influe sur le temps d'exécution des instructions au travers des politiques de remplacement des lignes dans le cache.

D'autre part, toujours à cause de la structure pipelinée des processeurs, les exécutions des blocs dans le processeur se chevauchent comme le montre la figure 2.6. Le pire temps d'exécution d'une séquence $[B_1..B_n]$ de blocs est plus courte que la somme des pires temps d'exécution des blocs qui la composent.

$$t_{B_1-\ldots-B_n} \leq \sum_{1 \leq i \leq n} t_{B_i}$$

Ainsi, la prise en compte des temps d'exécution sans tenir compte de leur recouvrement entraîne une surestimation du WCET global. Pour calculer un WCET plus précis, on utilise les **coûts** des blocs, c'est à dire le temps entre la fin de la dernière instruction du bloc précédent et la fin de la dernière instruction du bloc considéré. On a ainsi

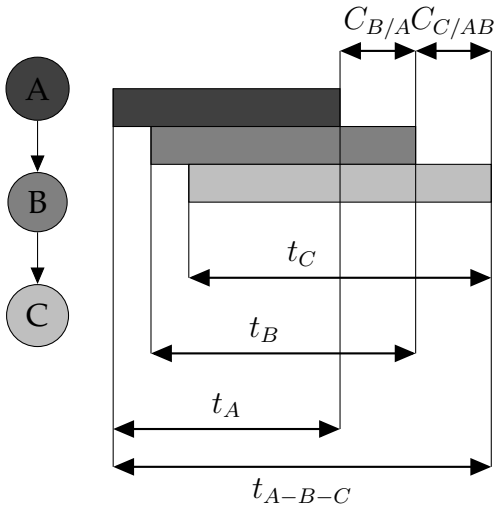


FIG. 2.6 – Le recouvrement des blocs de base

$$t_{B_1-...-B_n} = \sum_{1 \leq i \leq n} C_{B_i/B_1..B_{i-1}}$$

.

le temps d'exécution de la séquence $B_1 - ... - B_n$ est la somme des coûts des blocs de la séquence. Le coût C_{B_i} de chaque bloc B_i est calculé par rapport aux blocs qui le précède $B_1..B_{i-1}$.

Afin de d'obtenir des estimations de plus en plus proches du pire temps d'exécution réel, des méthodes ont été proposées pour la prise en compte des éléments architecturaux comme le pipeline d'exécution, les mémoires caches et le prédicteur de branchements.

2.1.3 Les méthodes hybrides

Ces méthodes font intervenir une partie statique et une partie dynamique. Généralement, il s'agit d'affiner, par des mesures ou des simulations, le calcul obtenu par analyse statique.

Ainsi, dans [66], les auteurs mesurent les temps d'exécution des branches de l'arbre syntaxique des programmes. Dans [125], la partie dynamique est prépondérante. L'analyse statique de parties du code est utilisée pour déterminer le prochain chemin à exécuter.

Les probabilités sont aussi utilisées pour affiner l'estimation du pire temps d'exécution [11, 12]. Dans un premier temps, des profils d'exécution des blocs de base sont dé-

finis en utilisant des mesures. Ces profils sont combinés pour obtenir un profil de chemin. C'est dans cette étape que les probabilités sont utilisées. L'étape suivante permet de prendre en compte les dépendances entre les profils de blocs de base. Le pire temps d'exécution calculé est probabiliste. L'objectif de la méthode est d'affiner la marge souvent ajoutée par les industriels, et non d'obtenir une borne supérieure du pire temps d'exécution.

D'autres études proposent de modifier la granularité de l'analyse. Dans [127], l'élément de base n'est pas le bloc de base mais une séquence d'instructions indépendantes du jeu de données en entrée du programme. Le calcul se fait par la technique d'énumération de chemins implicites en utilisant ces segments en lieu et place des blocs de base. Dans [97], Petters introduit les blocs de mesure qu'il définit en fonction du nombre de chemins qu'ils contiennent, dans le but de mesurer de manière déterministe le temps d'exécution de ces blocs. Le pire temps d'exécution global combine ces mesures.

2.2 Le parallélisme d'instructions

Dans la séquence d'instructions qui constitue une tâche, celles qui sont indépendantes peuvent être exécutées en parallèle. Il y a trois types de dépendances qui limitent ce parallélisme : les dépendances de données, les dépendances structurelles et les dépendances de contrôle.

Les **dépendances de données** sont de trois sortes :

- les **vraies dépendances** : une instruction utilise une donnée produite par une instruction qui la précède, elle doit attendre la production de cette donnée,
- les **antidépendances** : une instruction veut écrire dans un registre qu'une instruction précédente doit lire. L'écriture ne peut commencer avant la fin de la lecture qui la précède,
- les **dépendances de sortie** : une instruction veut écrire dans un registre qu'une instruction précédente doit également modifier. De même que dans le cas précédent, l'instruction doit attendre la libération du registre pour faire son travail.

Les **dépendances structurelles** ont lieu lorsque plusieurs instructions entrent en concurrence pour l'accès à une ressource du processeur. Cela peut être une unité fonctionnelle, un emplacement dans une file ou dans un tampon. Dans tous les cas, l'exécution est sérialisée au niveau de cette ressource.

Lorsqu'on rencontre un branchement, on doit déterminer s'il est pris (s'il est conditionnel) et vers quelle adresse cible il détourne le flot de contrôle du programme. Faute

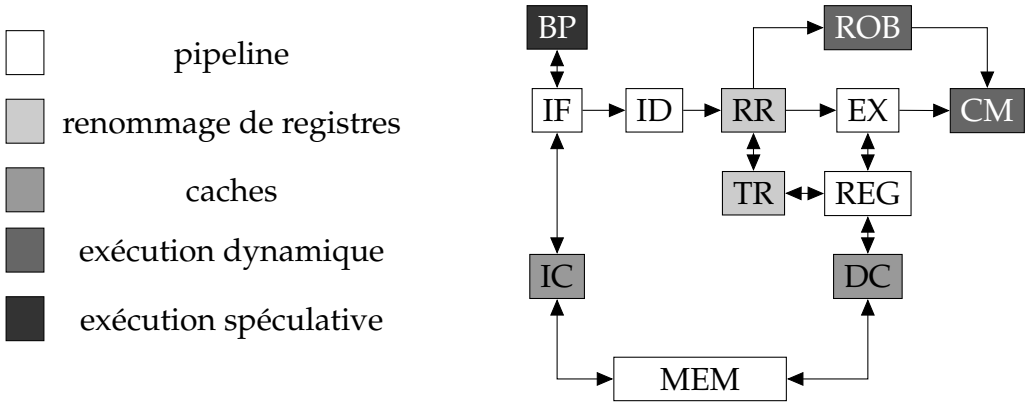


FIG. 2.7 – Processeur moderne

de savoir quelles instructions doivent être chargées, on ne peut pas continuer l'exécution. C'est une **dépendance de contrôle**.

2.2.1 Exploiter le parallélisme d'instructions

Le mécanisme de base des processeurs modernes est le **pipeline d'instructions**, qui permet un recouvrement partiel de l'exécution des instructions. L'exécution d'une instruction peut être décomposée en plusieurs étapes successives. On peut par exemple imaginer une décomposition en trois étapes comme dans la figure 2.7 :

- le chargement de l'instruction depuis la mémoire (IF),
- le décodage (ID) de l'instruction durant lequel on reconnaît l'instruction et ses paramètres,
- l'exécution proprement dite (EX).

Le pipeline d'instructions permet d'augmenter le débit du processeur. Le temps de cycle est le temps de traitement de la plus longue étape élémentaire. Le pipeline d'instructions permet donc de réduire le temps de cycle du processeur. De plus, plusieurs instructions sont traitées au même moment, chacune étant dans une étape élémentaire différente. Les dépendances limitent l'efficacité du pipeline en introduisant des bulles, c'est à dire des cycles pendant lesquels certains étages sont vides.

Des mécanismes architecturaux ont été mis en place pour pallier à de tels problèmes. Ils sont illustrés dans la figure 2.7.

Par exemple, un mécanisme de **renommage de registres** peut être utilisé pour limiter les dépendances de données aux dépendances vraies (lecture après écriture), les antidépendances et les dépendances de sortie étant totalement éliminées. On réduit ainsi le nombre de bulles, mais cela nécessite l'ajout d'un étage (RR) et la mise à jour

des tables de renommage (TR).

On peut aussi **ordonnancer dynamiquement** les instructions de manière à les exécuter dans un ordre différent de celui du programme tant que les dépendances vraies sont respectées (lorsqu'une instruction est bloquée en attente d'une donnée, les instructions placées derrière peuvent passer devant si elles ne sont pas dépendantes de l'instruction bloquée). Le maintien de la cohérence de l'état du processeur (en particulier lors du traitement d'une interruption) demande l'utilisation d'une file dite "tampon de réordonnancement" (ROB) et l'ajout d'un étage au pipeline (CM).

On peut aussi signaler que l'impact des dépendances de contrôle peut être limité par l'**exécution spéculative** d'une branche ou l'autre (si la destination du branchement peut être prédite). Cela nécessite de recourir à un prédicteur de branchements (BP) dont l'efficacité n'est pas absolue et dont les erreurs ont un coût : il faut vider le pipeline des instructions exécutées à tort, ce qui prend souvent plusieurs cycles, et l'état du pipeline doit redevenir ce qu'il était au moment du branchement, ce qui nécessite une sauvegarde et une restauration de certaines structures comme les tables de renommage.

Pour augmenter le parallélisme, on construit des processeurs **superscalaires**. C'est à dire que les structures d'action sont multipliées pour permettre le traitement de plusieurs instructions à chaque cycle. Par exemple, un processeur superscalaire à quatre voies permet de décoder et exécuter jusqu'à quatre instructions par cycle. Ce dispositif demande, en plus de l'ajout de structures d'action, un accroissement de la taille des structures de stockage sous peine de multiplier les dépendances structurelles. Il augmente aussi les pénalités de mauvaise prédiction dans le cadre de l'exécution spéculative car il faut purger davantage d'instructions du pipeline. La superscalarité rend aussi le pipeline plus sensible aux dépendances. Tout blocage d'un flot de contrôle, suite à une dépendance de donnée par exemple, induit la non utilisation de davantage de structures en aval dans le processeur et donc une moins bonne utilisation de ses ressources.

Pour atténuer l'effet des dépendances structurelles, il faut éviter que lorsqu'une instruction a besoin d'une structure d'action, celle-ci soit déjà occupée. Cette situation peut se produire dans deux cas :

- lorsque la structure d'action a une latence strictement supérieure à un cycle. Pour éviter cette situation, on peut pipeliner la structure d'action pour que chaque étage ait une latence d'un cycle. Ceci n'est pas toujours possible. Par exemple, les diviseurs sont difficiles à pipeliner.
- dans un processeur superscalaire, lorsque plusieurs instructions exécutées en

même temps ont besoin de la même structure et la capacité de cette structure est insuffisante pour les accueillir toutes. On peut être amené à revoir la capacité de cette structure pour éviter la contention. Cela peut amener à les surdimenssionner car ces contentions ne se produisent pas systématiquement.

Le recours à tous ces mécanismes améliore grandement les performances moyennes des processeurs modernes. Cependant, ils les rendent de plus en plus complexes, et donc de plus en plus coûteux, pour un gain qui devient de plus en plus faible à mesure que les techniques s'affinent [56]. Pour remplir les structures internes sous-exploitées dans les processeurs, on exploite maintenant le parallélisme de tâches, selon des techniques que nous aborderons un peu plus loin.

2.2.2 Parallélisme d'instructions et calcul de pire temps d'exécution

Avant l'apparition du pipeline d'instructions, on pouvait se contenter pour estimer le temps d'exécution d'un bloc, d'ajouter les temps d'exécution de ses instructions. Le recouvrement partiel des instructions induit par le pipeline rend cette approche grossière au sens où l'estimation obtenue est peu précise. Or une surestimation importante du pire temps d'exécution du programme n'est pas souhaitable car elle conduit à un surdimentionnement du système et donc à des surcoûts importants.

Prise en compte du pipeline d'instructions

Pour la prise en compte du pipeline d'instruction, une technique basée sur la représentation de l'état du pipeline par des tables de réservation a été proposée [63]. Cette méthode n'est efficace que pour un pipeline scalaire et n'est pas utilisable dans le cadre de l'ordonnancement dynamique ou de l'exécution spéculative.

Une autre solution proposée est d'évaluer, par interprétation abstraite, tous les états possibles du pipeline au début de l'exécution de chaque bloc de base [104]. Pour chacun de ces états, on peut calculer un coût d'exécution du bloc. La valeur maximale parmi tous les coûts possibles est le pire coût du bloc. L'outil aiT [1] utilise cette méthode.

Une autre méthode consiste à représenter l'exécution du bloc dans le pipeline par un graphe orienté. Ce graphe est appelé graphe d'exécution. La méthode a été initialement proposée par Li, Mitra et Roychoudoury de l'université de Singapour dans [73]. Elle permet de modéliser les pipelines scalaires à ordonnancement dynamique. Dans ces graphes, les nœuds représentent des couples (étage/instruction) et les arcs modélisent les liens de précédence entre les nœuds. L'analyse consiste à calculer les dates

relatives auxquelles les différents nœuds peuvent être exécutés en posant des hypothèses pessimistes sur le contexte d'exécution des blocs.

Nous avons étendu ce modèle pour permettre la prise en compte de pipelines superscalaires et d'unités fonctionnelles multiples [6]. Pour ce faire, il était nécessaire de prendre en compte l'exécution concurrente des instructions ainsi que la gestion des contentions lors de l'accès aux ressources.Nous avons introduit deux nouveaux types d'arcs :

- des arcs barrés qui relient deux nœuds qui peuvent être exécutés simultanément ou dans l'ordre du programme,
- des arcs pointillés non orientés et valués qui relient les nœuds qui peuvent entrer en conflit pour l'accès à une même ressource. La valeur associée à l'arc représente la capacité de la ressource pour laquelle les instructions peuvent être en contention.

Une nouvelle extension a été proposée dans [105]. Elle propose un autre algorithme de résolution qui permet d'affiner les coûts estimés en considérant l'impact des dates de libération de chacune des ressources nécessaires au bloc. Le coût de chaque bloc est exprimé en fonction de son contexte d'exécution. Le contexte est défini par un ensemble de paramètres qui sont les dates de mise à disposition de toutes les ressources nécessaires à l'exécution du bloc comme, par exemple, les étages du pipeline, les unités fonctionnelles, les valeurs des registres ...Le patron d'exécution du bloc est exprimé en fonction de ces paramètres et peut ensuite être analysé pour calculer une borne supérieure du pire temps d'exécution du bloc.

Le contexte d'exécution est défini comme l'ensemble des ressources qui, si elles ne sont pas disponibles lorsqu'elles sont requises, peuvent avoir un impact sur le temps d'exécution du bloc. Deux types de ressources peuvent être mis en évidence :

- les **ressources structurelles** comme les étages du pipeline, les unités fonctionnelles, les emplacements dans les files d'instructions. Une instruction du bloc peut être retardée si elle entre en compétition avec une autre instruction pour utiliser une ressource structurelle.
- les **ressources fonctionnelles** comme les valeurs des registres. Pour être exécutée par une unité fonctionnelle, une instruction doit attendre que ses opérandes soient prêts, ce qui peut être vu comme la mise à disposition de cette ressource par l'instruction qui la produit.

Pour l'analyse du coût d'exécution des blocs de base, on utilise les graphes d'exécution qui seront présentés plus en détails dans la section 3.2.

Les contraintes sur chaque nœud du graphe d'exécution sont exprimées au travers

de deux vecteurs ε et δ qui indiquent respectivement :

- la dépendance de la date à laquelle le nœud est prêt envers chaque ressource du contexte d'exécution,
- le délai minimal entre la disponibilité de la ressource et la date à laquelle le nœud est prêt.

Dans l'étape suivante, les dates auxquelles chacun des nœuds est prêt sont calculées. Les deux vecteurs sont d'abord initialisés : ε_N^r est vrai si le nœud N utilise la ressource r, et δ_N^r est initialisé à 0, ce qui signifie que le nœud N ne peut commencer que lorsque la ressource r est disponible. Puis le graphe est parcouru dans l'ordre topologique et les dépendances sont propagées de nœud en nœud. Un nœud dépend d'un autre si un de ses prédécesseurs en dépend. Les délais sont aussi propagés. Le délai relatif à une ressource est le plus grand délai des prédécesseurs du nœud majoré de la latence de la ressource.

Prise en compte de la prédiction de branchement

Pour ce qui est de la prédiction de branchements, les premiers travaux consistent à rechercher le nombre de mauvaises prédictions et d'appliquer au pire temps d'exécution de la tâche la pénalité de mauvaise prédiction pondérée par le nombre de mauvaises prédictions. Ils prennent en compte le délai maximum de mauvaise prédiction fourni par les constructeurs des processeurs [72, 28].

On peut affiner le coût de la mauvaise prédiction en mesurant le temps d'exécution des séquences de blocs (bloc contenant le branchement et ses successeurs) lorsque le branchement est mal prédit et lorsqu'il est correctement prédit. Les temps d'exécution des séquences son calculés dans le cas d'un bonne prédiction et dans le cas d'une mauvaise prédiction. La plus grande différence entre les deux est prise comme pénalité. Dans [17], les auteurs montrent que le coût d'une mauvaise prédiction dépend de la direction du branchement considéré.

Dans [15], l'auteur modifie le graphe de flot de contrôle pour prendre précisément en compte l'impact de la prédiction des branchements conditionnels. Les arêtes sortantes dun bloc qui se termine par un branchement conditionnel (pris et non pris), sont dédoublées. Il y a donc quatre arêtes : pris bien prédit, pris mal prédit, non pris bien prédit et non pris mal prédit.

La prédiction statique consiste à attribuer une prédiction fixe à chaque branchement. Comme il n'y a que deux arêtes (dans le graphe de flot de contrôle) en sortie du bloc de base qui contient le branchement, une fois que les temps d'exécution respectifs qui y sont associés sont déterminés, le calcul est identique à celui effectué sans

prédiction de branchement [15].

Pour ce qui concerne la prédiction dynamique, elle évolue, pour un même branchement, au cours de l'exécution du programme. On a donc deux critères à prendre en compte : le branchement est pris ou non pris, bien prédit ou mal prédit, soit en tout quatre combinaisons. Les premiers modèles qui ont pris en compte la prédiction dynamique de branchements pour l'estimation du pire temps d'exécution des programmes considéraient tous les branchements comme mal prédits, ce qui a pour conséquence une surévaluation de l'impact de la prédiction de branchement.

Dans la prédiction de branchement, il faut distinguer deux aspects : la prédiction de la direction du branchement et la prédiction de l'adresse cible du branchement. La prédiction de l'adresse cible est faite par la table des cibles de branchements (BTB[20]). Cette table mémorise les adresses cibles des branchements déja rencontrés. Des confilts d'accès, et donc des mauvaises prédictions de la cible du branchement, sont possibles. Pour modéliser les conflits d'accès à la table des cibles de branchements, Colin et Puaut ont utilisé la simulation statique [28]. Ils calculent le contenu de la table avant et après chaque branchement et classent les branchements en fonction des conflits qui peuvent survenir :

- *toujours D-prédits* : l'accès à la table des cibles de branchements pour les branchements de cette catégorie est toujours un échec.
- *toujours inconnu* : il n'est pas possible de prédire le succès ou l'échec d'accès à la table des cibles de branchements pour cette catégorie.
- *premier D-prédit* : le premier accès à la BTB est un échec, les autres accès sont des succès.
- *premier inconnu* : on ne sais pas si la prédiction sera bonne pour le premier accès, les accès suivants sont des succès.

Une fois les conflits détectés, chaque occurence de branchement est traité en fonction de sa catégorie et des conflits détectés.

L'autre problématique de la prédiction de branchement consiste à déterminer si un branchement conditionnel est pris ou pas. Il existe plusieurs sortes de prédicteurs pour réaliser cette tâche. Pour les modéliser, deux approches existent :

- l'approche locale consiste à analyser chaque branchement selon la structure algorithmique à laquelle il est associé.
- l'approche globale utilise une modélisation générique des branchements.

Dans l'analyse locale, le calcul du nombre de mauvaises prédictions le long du chemin pire cas peut se faire de plusieurs manières. Colin et Puaut [28] le calculent en fonc-

[20] Branch Target Buffer

tion de la structure algorithmique associée au branchement. Bate et Reutemann [9] affinent ce calcul en se focalisant sur les branchements associés aux structures algorithmiques conditionelles (si-alors-sinon) et itératives (boucles) dont le nombre d'itérations est fixe. Leurs travaux ont ensuite été étendus pour prendre en compte une pénalité de mauvaise prédiction plus précise et un nombre d'itérations variable pour les boucles [17, 16].

L'analyse globale modélise le prédicteur pour anticiper son évolution et donc prévoir le nombre de prédictions correctes. Cette analyse utilise la méthode d'énumération des chemins implicites. Li et al. ont d'abord décrit un prédicteur fonctionnant avec un historique global et un compteur 1 bit [72]. Ils prennent en compte les conflits possibles sur les entrées de la table contenant les compteurs. Burguière et Rochange ont ensuite décrit l'évolution d'un compteur 2 bits [18]. Puis elles ont proposé la modélisation de prédicteurs globaux 2 bits et locaux 2 bits [19].

2.3 La hiérarchie mémoire

Le temps d'accès à la mémoire pose des problèmes de performances dans les systèmes informatiques. En effet, plusieurs ordres de grandeur peuvent séparer le temps de cycle du processeur du temps d'accès à la mémoire. Pendant l'accès mémoire, dans le meilleur des cas, toutes les instructions qui dépendent directement ou indirectement de cet accès sont bloquées. A ceci, il faut ajouter les blocages liés à la saturation des éléments de mémorisation en amont, à cause de la présence des instructions dépendantes. Dans le cas d'un processeur qui exécute les instructions dans l'ordre du programme, à chaque accès mémoire, c'est tout le pipeline qui est bloqué en attente de la résolution de l'accès.

Pour compenser partiellement cet écart de performance, on met en place une hiérarchie mémoire. Les mémoires rapides ne peuvent pas être de grande capacité (et sont plus chères). L'idée est d'insérer une mémoire de capacité intermédiaire (plus rapide) entre la mémoire centrale et le processeur. Ces mémoires intermédiaires sont appelées des **mémoires caches**. On peut utiliser plusieurs niveaux de caches, de plus en plus petits et rapides à mesure qu'on se rapproche du processeur, d'ou l'appellation de hiérarchie.

L'usage d'un tel mécanisme est justifié par le principe de localité : un programme n'accède pas aux instructions et aux données de manière uniforme, mais a tendance à réutiliser des données et des instructions qu'ils ont utilisé recemment. On observe deux types de localités :

– la **localité temporelle** : les données ou instructions qui ont été accédées recemment ont de grandes chances de l'être dans un futur proche. Le corps d'une boucle illustre ce type de localité.
– la **localité spatiale** : les éléments dont les adresses sont proches d'un élément référencé auront tendance à être référencés dans un futur proche. Les tableaux sont un exemple de structure qui favorise la localité spatiale.

2.3.1 Le principe des mémoires caches

Dans une hiérarchie mémoire, les caches sont souvent inclusifs, c'est à dire que les données présentes dans le cache de premier niveau (le plus proche du processeur) sont aussi présentes dans le cache de second niveau et ainsi de suite jusqu'à la mémoire centrale. Les processeurs actuels contiennent souvent 3 niveaux de cache.

Le fonctionnement est le suivant : lorsque le processeur requiert une donnée, si celle-ci est dans le cache de premier niveau, elle est directement chargée vers le processeur. Si elle n'y est pas, sa présence est recherchée dans l'élément suivant de la hiérarchie jusqu'à ce qu'elle soit trouvée. En pratique, pour des raisons de performances, la recherche se fait souvent en parallèle.

Il existe plusieurs politique d'accès aux caches :

– La politique d'accès direct dans laquelle la mémoire est divisée en plis de la taille du cache. Chaque pli est découpé en blocs qui correspondent aux lignes du cache. Chaque bloc n de chaque pli correspond à la ligne n du cache.
– la politique associative : chaque bloc peut aller dans n'import quelle ligne du cache.
– la politique associative par ensembles : c'est un cache a accès direct dans lequel chaque ligne est associative et peut donc contenir des blocs issus de plis différents.

Dans les deux derniers types, il faut décider d'une politique de remplacement des blocs dans le cache. Les plus utilisées sont LRU[21] dans laquelle on remplace le bloc le moins recemment utilisé, FIFO[22], dans laquelle le premier bloc entré dans la ligne de cache est remplacé. On peut aussi remplacer un bloc au hasard.

Lorsqu'on écrit dans un cache, on peut écrire aussi immédiatement dans le niveau inférieur, mais on peut aussi différer l'écriture pour des raisons de performance. L'écriture se fait alors lors du remplacement du bloc.

[21] Least Recently Used
[22] First In First Out

Les caches unifiés et les caches séparés

Lorsqu'on utilise des caches différents pour les instructions et pour les données, on parle de caches séparés. Dans le cas contraire, on parle d'un cache unifié. L'intérêt de la séparation des caches est de permettre l'accès simultané aux données et aux instructions. De plus, comme les instructions et les données ont des comportements différents, la séparation des caches permet une analyse plus fine lors de l'estimation du pire temps d'exécution comme nous le verrons dans la section 2.3.2.

Les mémoires scratchpads

L'utilisation des mémoires caches est transparente pour le programmeur. Il n'en est pas de même des mémoires scratchpads. Ce sont des mémoires rapides et de petite taille qui sont gérées par logiciel. Le programmeur choisit de charger une partie des données dans cette mémoire pour en accélérer l'accès. Contrairement à la mémoire cache, la mémoire scratchpad est incluse dans l'espace d'adressage du processeur.

Les mémoires caches et scratchpads peuvent coexister dans un système informatique. La principale différence entre elles réside en ce que la mémoire scratchpad garantit un temps d'accès fixe aux données qu'elle contient alors que le temps d'accès à une donnée au travers d'une hiérarchie mémoire varie en fonction des succès et des échecs lors de l'accès aux caches. Plus la donnée recherchée sera dans un niveau de cache proche du processeur, plus l'accès sera rapide.

2.3.2 La prise en compte de la hiérarchie mémoire dans le calcul du pire temps d'exécution

Les mémoires caches ont un impact important sur le temps d'exécution du programme. Un défaut de cache génère une latence importante car le temps d'accès à la mémoire est beaucoup plus grand que le temps d'accès au cache par rapport au temps de cycle d'un processeur moderne. On trouve deux grandes classes de méthodes qui permettent la prise en compte des caches dans le calcul du pire temps d'exécution.

La première est basée sur une catégorisation des accès en fonction de leur présence potentielle dans le cache au moment de l'accès. Les catégories sont :

- "*always hit*" lorsque l'accès est toujours un succès,
- "*always miss*" regroupe les accès pour lesquels la donnée n'est jamais présente dans le cache au moment de l'accès,

- "*conflict*" est la catégorie dans laquelle on met les accès pour lesquels on ne sait pas prédire le succès ou l'échec,
- "*first miss*" regroupe les accès dont la première occurrence est un échec mais les suivantes sont toutes des succès. Cette catégorie n'est pas toujours présente. Elle correspond par exemple aux corps des boucles, souvent nombreuses dans les programmes, qui sont absents du cache à la première itération mais présents lors des itérations suivantes.

Cette catégorisation peut s'obtenir par simulation statique [86, 54, 114]. Le nombre d'états possibles du cache est alors réduit grâce aux informations obtenues lors de la compilation, ou par interprétation abstraite [86, 42, 116, 96].

Le second type de méthode consiste à évaluer le contenu du cache. Le cas du cache d'instructions est le plus simple. Le graphe de conflits de cache (CCG[23]) a été introduit par Li et al. dans [118]. Un l-block est une séquence d'instructions qui appartiennent au même bloc de base et qui sont contenues dans la même ligne de cache. Un CCG est généré pour chaque ligne de cache et permet de décrire l'ordre d'exécution des l-blocks de la ligne. On exprime ensuite ce graphe sous forme d'un système de contraintes dont la résolution permet de connaître le nombre d'échecs d'accès au cache.

Le problème est plus complexe avec un cache de données car les adresses sont souvent calculées dynamiquement par le programme. Il est donc nécessaire d'effectuer une phase d'analyse pour déterminer les adresses avant de modéliser les accès. Ce calcul peut se faire pendant l'analyse de flots. Deux états sont générés pour chaque donnée car chaque donnée peut être lue ou écrite en mémoire [118]. Il est aussi possible d'utiliser l'interprétation abstraite [43] ou la simulation statique [63] pour l'étape de calcul des adresses des données. Toutes les valeurs possibles sont alors calculées.

Le gel du cache a aussi été proposé. Il consiste à figer le cache pour de petites portions du code [122] ou pour le programme entier [99, 4]. Le choix des instructions et des données à cacher est fait par logiciel. Cette méthode est déterministe, elle n'entraîne pas de surestimation du pire temps d'exécution. Le cache se comporte alors comme un scratchpad.

2.4 Le parallélisme de tâches

Le mécanisme de base pour l'exploitation du parallélisme de tâches est le **multiprocesseur** (MIMD[24]), plusieurs processeurs exécutent chacun une tâche. La synchro-

[23] Cache Conflict Graph
[24] Multiple Instructions, Multiple Data

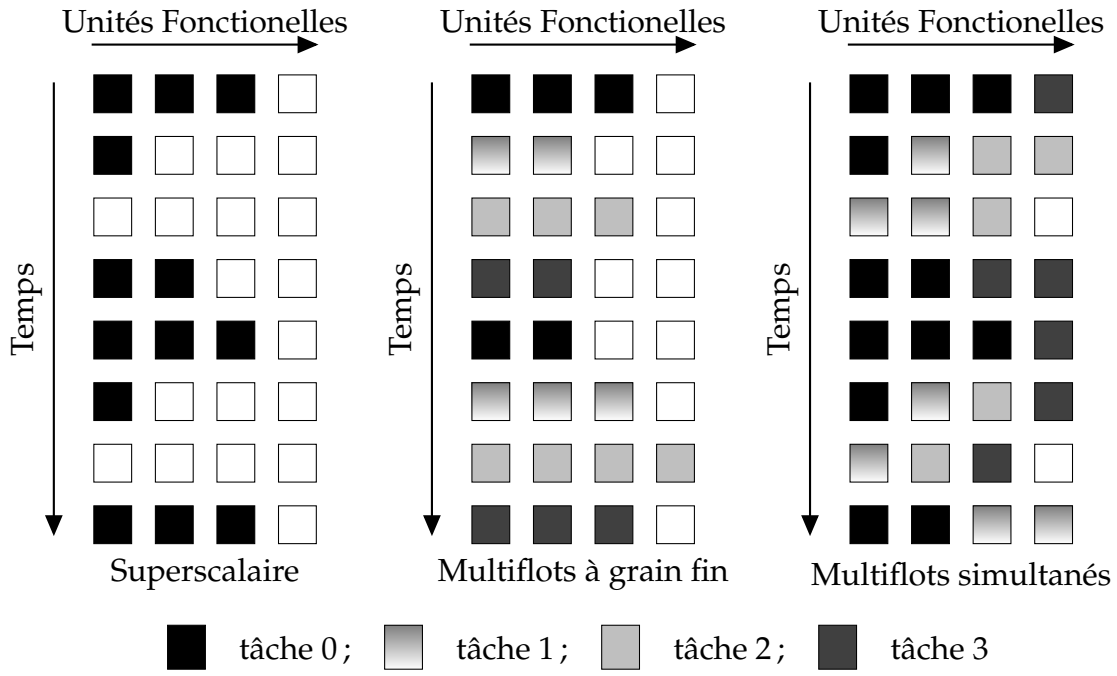


FIG. 2.8 – Les avantages des processeurs multiflots

nisation entre les tâches peut se faire de deux manières : par *partage de la mémoire* entre les processeurs ou bien *par messages*. Dans les deux cas, un réseau d'interconnexion est nécessaire. Il est d'autant plus complexe que le nombre de processeurs est grand. Les processeurs peuvent être embarqués sur un même circuit intégré, on parle alors de **processeur multicœurs**. Les cœurs partagent leurs structures externes, comme le bus mémoire, mais aussi des structures embarquées dans la puce comme le cache de niveau 2. Ce sont des multiprocesseurs à mémoire partagée le plus souvent.

Afin d'optimiser l'utilisation des structures internes du processeur lorsque le parallélisme d'instructions n'est pas suffisant, et d'éviter les bulles dans le pipeline et d'augmenter le débit du processeur, les processeurs multiflots sont composés d'un seul cœur dans lequel les structures internes sont partagées entre les tâches. On trouve aussi des combinaisons de ces deux mécanismes : des processeurs multicœurs dans lesquels chaque cœur est multiflot.

Les processeurs multiflots permettent d'exécuter plusieurs flots de contrôle. On distingue les **multiflots à grain fin**, dans lesquels l'exécution se fait en alternance des **multiflots simultanés** dans lesquels les différents flots de contrôle peuvent être exécutés simultanément.

Les avantages des processeurs multiflots sont les suivants.

– dans un processeur superscalaire, lorsque le parallélisme d'instruction est in-

suffisant, toutes les unités fonctionnelles sont inoccupées pendant un ou plusieurs cycles, le plus souvent à cause d'opérations à latence longue. C'est la **sous-utilisation verticale** identifiée dans [34]. Les processeurs multiflots à grain fin limitent ce type de sous-utilisation en permettant l'exécution d'instructions venant d'un autre flot de contrôle pendant ces cycles.
– la **sous-utilisation horizontale** se produit quand certaines unités fonctionnelles sont inutilisées. Ce sont les processeurs multiflots simultanés qui permettent de limiter cette sous-utilisation en autorisant l'utilisation de ces unités par des instructions d'un autre flot de contrôle.

2.4.1 Les processeurs multiflots simultanés

La particularité des **processeurs multiflots simultanés** (SMT[25]) introduits par Tulsen *et al* [119] est *le partage du processeur par plusieurs tâches*. Leur fonctionnement est similaire à celui d'un processeur classique. Un tel partage permet de limiter la sous-utilisation des structures du processeur, comme l'illustre la figure 2.8. Dans ces processeurs, certaines structures sont dupliquées, comme le compteur de programme ou la table de renommage, d'autres sont partagées, comme les caches ou les unités fonctionnelles. Ce partage est peu coûteux en transistors : pour traiter deux tâches, le Pentium 4 Hyper Threading d'Intel ne demande que 5 % d'augmentation de taille de circuit [82].

Pour une ressource de stockage, c'est la quantité de données que peut contenir la ressource qui est partagé, nous appellerons donc le partage de ce type de ressources **partage spatial**.

Lorsque, dans le pipeline, on réduit le degré de parallélisme, on a une *sérialisation* de l'exécution qui nécessite de choisir dans quel ordre les instructions qui arrivent en parallèle sont traitées. Ce choix déterminant le moment auquel chaque instruction disposera de la ressource, ce partage est dit **temporel**.

Il existe plusieurs politiques de partage spatial des ressources :
– le **partage statique** dans lequel la ressource partagée est divisée en autant de parties que le processeur peut exécuter de tâches simultanément. Comme le montre la figure 2.9(a), chaque partie est réservée à une tâche,
– le **partage dynamique** dans lequel aucun contrôle n'est fait sur l'équité de la répartition de la structure comme le montre la figure 2.9(b). La ressource peut donc être totalement occupée par une seule tâche,

[25]Simultaneous MultiThreading

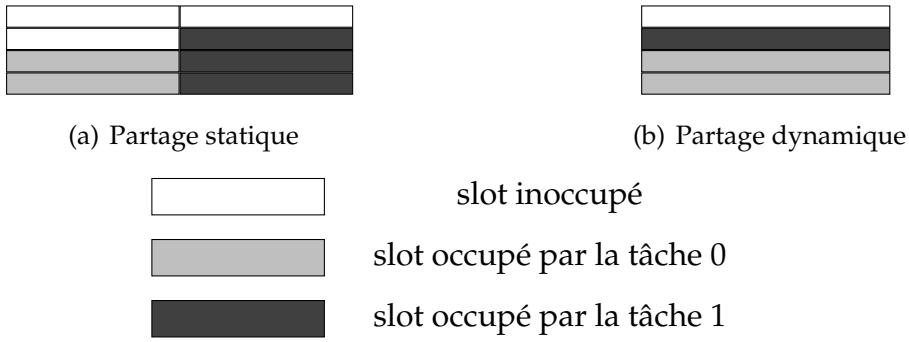


FIG. 2.9 – Partage des structures de stockage

– le **partage dynamique avec maximum** qui ressemble au précédent à ceci près qu'une tâche doit toujours laisser une partie de la structure disponible pour les autres. Cette politique permet à une tâche de s'exécuter plus rapidement (en utilisant plus de ressources) lorsque l'autre est bloquée. Elle évite cependant les famines en empêchant une tâche de monopoliser la totalité d'une ressource.

Dans les processeurs multiflots simultanés existants, plusieurs politiques de partage temporel ont été implantées :

– la politique du **tourniquet** (RR[26]) consiste à ce que chaque tâche soit traitée à son tour, circulairement. Une version optimisée existe, qui saute le tour des tâches bloquées.
– dans la politique **parallèle**, toutes les tâches sont traitées simultanément à chaque cycle.
– dans la politique préemptive à priorité fixe, une priorité est associée à chaque tâche. La tâche de plus haute priorité est préférée, à chaque fois que possible, c'est à dire à chaque cycle, sauf si elle est bloquée.
– dans **ICOUNT** [22, 24, 119], la priorité de chaque tâche est calculée dynamiquement, à chaque cycle, en fonction du nombre d'instructions de la tâche qui ne sont pas prêtes et qui risquent de rester longtemps dans le pipeline. La priorité de chaque tâche est inversement proportionelle à son potentiel de blocage du pipeline. Certaines implantations de cette politique permettent de traiter des instructions venant de plusieurs tâches en même temps (les plus prioritaires).

D'autres politiques similaires à ICOUNT ont été proposées, parmi lesquelles on peut citer **BRCOUNT** [23] qui considère le nombre de branchements dans les étages de pré-émission et **DCRA** [120] qui est fondée sur l'usage des ressources.

[26]Round Robin

Peu de données sont disponibles au sujet des implantations existantes de la technologie multiflots simultanés.L'*Alpha 21464* [31, 98, 32] de Compaq n'est pas arrivé à la phase de production (2001). Il aurait pu exécuter jusqu'à quatre tâches simultanément et devait privilégier le partage dynamique. Seuls les tables de renommage et les registres devaient être partagés statiquement.

Le *Pentium 4 Hyper-Threading* [82] d'Intel, commecialisé dès 2002, peut exécuter jusqu'à deux tâches. Dans ce processeur, c'est le partage statique qui est privilégié. Au niveau de l'exécution, le choix des instructions lancées repose sur une politique de tourniquet.

Les *POWER5* [60, 83, 111] et *POWER6* [71] d'IBM sont sortis respectivement en 2004 et 2007. Le POWER5 comporte deux cœurs multiflots simultanés qui peuvent exécuter chacun deux tâches. Les instructions lancées sont choisies par une politique de tourniquet. Jusqu'à huit instructions provenant du même flot de contrôle peuvent être lancées en même temps. Elles sont ensuite traitées par groupes de cinq (consécutives et issues de la même tâche). Le retrait se fait selon la politique d'ordonnancement parallèle. Le processeur gère des priorités pour les tâches (de zéro à sept, sept étant la priorité la plus forte). La priorité change dynamiquement en fonction de l'occupation du pipeline par chaque tâche. La priorité des tâches qui utilisent le plus le pipeline est baissée. L'objectif est de limiter l'engorgement du processeur par les instructions d'une tâche et d'équilibrer le partage du processeur entre les tâches. La prise en compte de la priorité se fait au décodage. Le nombre de cycles de décodage alloués à la tache est proportionnel à sa priorité. On peut aussi attribuer les priorités de manière logicielle. Un mode "économie d'énergie" est activé lorsque toutes les tâches ont une priorité inférieure ou égale à un. Dans ce mode, la politique de partage temporel à la sortie des files d'instructions est semblable à ICOUNT.

Le POWER6 est aussi un processeur double cœur dont chaque cœur est multiflots simultanés à deux voies. Il est plus simple que son prédécesseur pour permettre une augmentation significative de la fréquence et une baisse de la consommation d'énergie. La logique de décodage est dupliquée.

Enfin Intel a commercialisé récemment deux processeurs multiflots simultanés :

– l'*Atom* [51], un processeur multiflots simultanés à deux voies conçu pour l'informatique embarquée. C'est un processeur 64 bits qui consomme peu d'énergie. Il se compose de deux pipelines scalaires. L'ordonnancement n'est pas dynamique. Avec le peu d'informations qu'on a sur ce processeur, on peut penser que le partage statique a été privilégié et que l'accès aux étages est ordonnancé par la politique du tourniquet.

– le nehalem (i7) [110], un processeur multiflots simultanés à deux voies proche du pentium 4. Les tampons de chargement et de rangement en mémoire ainsi que le tampon de réordonnancement sont partagés statiquement (64 entrées par tâche). Les stations de réservation ainsi que les caches sont partagés dynamiquement.

Les problèmes de modélisation

Le manque de prévisibilité se manifeste dans les structures partagées. Les conflits potentiels d'accès à une ressource partagée entre les tâches induit la méconnaissance de la disponibilité de cette ressource. On ne peut connaître précisément le délai d'accès à une ressource dont on ne sait pas si elle est libre ou non. La manière dont la ressource est partagée conditionne la précision de la prédiction.

Le type de ressource (de traitement ou de stockage) et donc du type de partage dépendent des conflits qui ont lieu.

Le partage spatial Pour une structure partagée statiquement, on connaît toujours son état de remplissage. Il n'y a pas de conflits possible pour l'accès à la ressource. Cependant, le partage statique des structures annule une partie du bénéfice du SMT dont l'objectif est d'optimiser l'utilisation des structures partagées, en ne permettant pas à une tâche d'utiliser une partie vide de la structure si elle ne lui est pas allouée. De plus, cette solution reporte le problème sur le partage temporel qui doit être mis en œuvre à la sortie de la structure. Il faut en effet choisir de quelle tâche les instructions sont extraites.

Pour une structure partagée dynamiquement, les instructions sont extraites dans l'ordre de leur entrée, indépendamment de la tâche à laquelle elles appartiennent. Un tel partage est un point de sérialisation si l'exécution en amont dans le pipeline est parallèle. Il faut alors décider comment se fait la sérialisation, c'est à dire dans quel ordre les informations entrent dans la structure. En cas de débordement de capacité, cela va déterminer dans quel ordre les tâches seront bloquées. C'est donc le partage temporel à l'entrée de la structure qui est déterminant dans ce cas.

Le problème du partage dynamique avec maximum est le même que le précédent, seul le seuil est décalé. Cette politique est modélisable en considérant qu'une tâche a toujours la place minimale dans la structure afin de garantir que le temps d'exécution évalué sera sûr, c'est à dire supérieur au WCET réel. En pratique ce n'est pas toujours le cas, une telle modélisation entraîne donc des surestimations.

Dans les trois cas, une modélisation est possible pour calculer un pire temps d'exécution sûr, c'est à dire qui ne sous-estime pas le pire temps d'exécution réel. Pour le

partages dynamiques et dynamique avec maximum, la précision de la modélisation est perdue. Le partage statique permet en revanche d'obtenir des pire temps d'exécution plus proches du pire temps d'exécution réel.

Le partage temporel La politique préemptive à priorité fixe permet d'assurer qu'une tâche sera toujours privilégiée. Cela convient lorsqu'on a une seule tache temps réel et en acceptant d'affamer éventuellement les autres tâches.

On peut utiliser une politique d'ordonnancement par tourniquet. Elle a pour avantage d'assurer une certaine équité entre les tâches. On peut prévoir facilement le délai maximum avant que ce soit de nouveau le tour d'une tâche donnée (c'est le nombre de tâches exécutées). L'inconvénient est qu'on ne peut pas affiner cette valeur. Lorsqu'une tâche doit sauter son tour, une autre peut prendre sa place ce qui entraîne un décalage et donc une surestimation du temps du bloc. De telles situations peuvent se produire souvent ce qui rend la surestimation non négligeable.

La politique I-Count se base sur la proportion d'instructions de chaque tâche déjà présentes dans le pipeline. On ne peut donc pas savoir quelle tâche verra ses instructions extraites de la structure sans faire des hypothèse sur les autres tâches. Pour cette raison, et même si elle permet une meilleure équité dans l'utilisation des ressources par les tâches, cette politique ne convient pas au temps réel strict car elle ne permet pas la prévisibilité.

La politique qui consiste à traiter en parallèle une instruction de chaque tâche présente dans la structure partagée statiquement est totalement prévisible mais risque de provoquer un engorgement des structures qui suivent dans le pipeline car elle augmente le débit de l'étage. Cette politique est à réserver, comme dans le power5 par exemple, au dernier étage du pipeline.

On peut penser à d'autres politiques connues dans les ordonnanceurs des systèmes d'exploitation temps réels comme LLF[27] qui privilégie la tâche pour laquelle il reste le plus de travail en termes de temps d'exécution, ou EDF[28] qui privilégie la tâche dont l'échéance est la plus proche. Ces méthodes font appel à des algorithmes souvent complexes et leur utilisation pourrait compromettre le temps de cycle. De plus, ces algorithmes ont souvent besoin du WCET de la tâche pour calculer l'ordonnancement. Déterminer le WCET en présumant de sa valeur dans la modélisation du processeur nous paraît hasardeux.

[27] Least Laxity First
[28] Earliest Deadline First

Les processeurs multiflots simultanés et le temps réel

Des tentatives ont été faites pour la prise en compte de l'entrelacement des tâches à plusieurs niveaux. Une extension de la technique par énumération des chemins implicites a été proposée dans [31]. Les auteurs proposent de prendre en compte tous les entrelacements possibles de tâches ce qui conduit à des systèmes de contraintes qui peuvent être très grands et dont les temps de calcul sont rédhibitoires.

Lo et al. proposent, dans [109], l'adaptation des stratégies d'ordonnancement des tâches pour les rendre plus prévisibles, mais ils ne prennent pas en compte les interactions entre les tâches à l'intérieur du pipeline d'instructions.

Dans [61], Kato et al. introduisent le temps d'exécution multi-cas (MCET[29]). Il est calculé à partir de plusieurs pires temps d'exécution obtenus lors de l'exécution de la tâche considérée en concurrence avec plusieurs autres. Les auteurs ne considèrent que les tâches périodiques et sans dépendances. L'ensemble de tâches est défini au départ et ne peut être modifié. Des préemptions entre les tâches peuvent se produire à tout moment. Les temps d'exécution sont relevés sur une hyper-période, une moyenne des pires temps d'exécution est faite sur l'ensemble des exécutions de chaque tâche. Le temps d'exécution multi-cas est ensuite utilisé dans un ordonnancement de type EDF à la place du pire temps d'exécution. La complexité des chemins d'exécution des tâches considérées peut induire la nécessité de faire un grand nombre de mesures de pire temps d'exécution. L'application de cette méthode semble donc conditionnée à l'usage d'une architecture prévisible.

L'isolement temporel d'une tâche temps réel d'autres tâches qui n'ont pas de contrainte temporelle, est étudié dans [33]. Les auteurs proposent de préserver autant que possible la performance de la tâche temps réel mais la prévisibilité n'est pas assurée. L'isolement recherché n'est donc pas obtenu. Cette technique n'est donc pas adaptée au temps réel strict.

Carzola et al. ont proposé la garantie d'une certaine qualité de service pour un ensemble de tâches temps réel grâce à des mécanismes architecturaux [22, 21, 24]. Cette solution concerne le temps réel souple et sort donc du cadre de notre étude.

A un grain plus fin, on peut partager des ressources du processeur [103]. Les ressources de stockage ont été étudiées, particulièrement au travers de l'étude des caches. Le problème est le suivant : lorsque le cache est partagé dynamiquement, une tâche peut écraser la ligne d'une autre tâche, provoquant pour cette dernière un échec qui ne se serait pas produit si elle s'était exécutée seule. La taille de cache allouée à chaque

[29] Multi-Case Execution Time

tâche peut aussi varier dynamiquement suivant le comportement et les besoins des tâches. Ce comportement, s'il favorise les performances globales, induit de l'indéterminisme. La présence des données dans le cache pouvant dépendre des tâches non critiques, celle des tâches critiques ne peut être prédite de manière déterministe. Ces techniques ne sont donc pas adaptées au temps réel strict. Le partage statique des caches a donc été proposé [76, 108].

Dans une approche plus globale, Barre [5], propose une adaptation de l'architecture du processeur qui permet l'indépendance des flots d'exécutions. Son approche est basée sur les politiques de partages des ressources, qu'il choisit statiques pour les ressources de stockage. Il introduit une nouvelle politique pour le partage temporel des ressources de traitement : le partage de bande passante(BP[30]), qui permet un partage équitable de la bande passante des ressources entre les tâches critiques tandis que les tâches non critiques profitent de la bande passante non utilisée par les tâches critiques. Le calcul du pire temps d'exécution des tâches critiques se fait alors comme pour une tâche seule. Malgré un affinage de la modélisation des conflits de ressources internes du processeur, la surestimation est relativement importante et augmente avec le nombre de tâches co-exécutées. De plus, les mécanismes utilisés pour obtenir la prévisibilité induisent une rigidité à cause de la rigueur de la politique de partage de bande passante.

2.4.2 Les processeurs multicœurs

On entand par architecture multicœur, une architecture composée de plusieurs processeurs (cœurs) sur une même puce, reliés par un réseau d'interconnexion et à mémoire partagée. Il ne faut pas les confondre avec les systèmes sur puces (SoC[31]) qui embarquent tout le système ; En particulier, la puce d'un processeur multicœur n'embarque pas la mémoire centrale.

Chaque cœur a ses caractéristiques propres indépendamment des autres, ce qui permet de les classer en 2 catégories :

- les multicœurs homogènes dans lesquels tous les cœurs sont identiques,
- les multicœurs hétérogènes dans lesquels les cœurs sont différents (un DSP et un microcontrôleur par exemple).

Chaque cœur peut bénéficier des différentes technologies caractéristiques des processeurs modernes, en particulier, ils peuvent être SMT ce qui démultiplie le nombre de

[30] Bandwidth Partitioning
[31] System on Chip

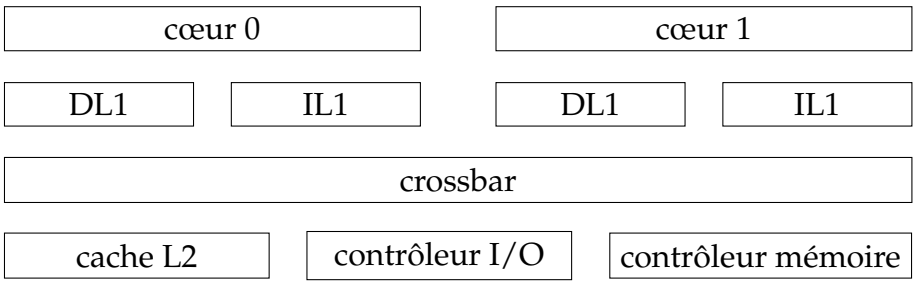


FIG. 2.10 – Architecture type d'un processeur bi-cœur homogène

tâches traitables simultanément par le processeur.

L'idée de base du modèle multicœur est de permettre l'exploitation du parallélisme de tâches. Cela permet de pallier le manque de parallélisme d'instructions de certaines applications. On atteint alors les mêmes performances qu'un processeur superscalaire monotâche [90]. Lorsque les deux types de parallélismes sont présents, on obtient même de meilleures performances. De plus, l'utilisation d'une seule puce limite l'augmentation de la consommation d'énergie.

Architecture générale

Comme le montre la figure 2.10, les processeurs multicœurs s'articulent autour d'un réseau qui relie les cœurs avec la hiérarchie mémoire et le système d'entrées/sorties. Ce réseau est souvent un bus ou un crossbar [70]. Les cœurs partagent donc certaines structures : le réseau, les caches (L2 et L3 le plus souvent) et l'accès à l'extérieur de la puce (mémoire, E/S...). Pour le reste, chaque cœur a ses structures propres, qui ne sont pas partageables avec les autre cœurs. Certaines architectures prévoient un module de communication avec d'autres modules du même type pour permettre une certaine extensibilité [7].

Caractéristiques particulières

Les processeurs multicœurs sont des multiprocesseurs mais le fait que les cœurs soient sur une même puce permet des partages qui sont impossibles aux multiprocesseurs "classiques".

Le partage des caches Si le partage de certaines structures comme le réseau d'interconnexion ou le bus de sortie de la puce n'est pas sujet à discussion, il n'en est pas de même d'autres éléments comme les caches. En effet, plusieurs politiques de partage

sont possibles, qui se situent à divers points d'équilibre entre simplicité d'implantation et performances [69], face à des problèmes comme la cohérence ou la conscistance. Des problèmes de passage à l'échelle s'ajoutent lorsqu'on pense à l'évolution des multicœurs en terme de multiplication du nombre de cœurs comme SPARC avec le *ROCK* qui comporte 16 cœurs dont chacun est multiflot simultané à deux voies. Chaque puce peut donc exécuter jusqu'à 32 tâches en parallèle.

Globalement, dans l'évolution des premiers multicœurs jusqu'à aujourd'hui, on assiste à un glissement de la ligne de partage : d'abord, seule la bande passante vers l'extérieur de la puce était partagée ; on a ensuite partagé le cache L2 puis on a rendu les cœurs SMT ce qui permet aux tâches de chaque cœur de partager le cache L1. Ce partage, de plus en plus proche des cœurs, diminue les latences lors des communications entre les tâches par variables partagées.

Conception du processeur Le nombre de cœurs et la largeur du pipeline (lancement des instructions) [35] sont aussi sujets à discussion. Le but de tels réglages est l'optimisation des performances. Ces choix de conception sont directement dépendants du rapport ILP/TLP des applications exécutées, et doivent donc êtres faits en fonction du type d'applications cibles. La tendance est à la multiplication du nombre de tâches exécutables en parallèles (TLP) sur des cœurs exploitant moins d'ILP. Les cœurs sont souvent plus simples : l'exécution n'est que rarement spéculative, elle se fait, en général, dans l'ordre et de façon scalaire. Ce gain en simplicité se traduit en un gain de transistors. Celui-ci est utilisé pour la multiplication des cœurs. La simplicité apporte aussi un gain de puissance dissipée et de consommation d'énergie, deux facteurs très appréciés dans le domaine des sysèmes embarqués. Enfin la simplicité se traduit généralement par une réduction des coûts de conception et de production.

Les systèmes d'exploitation et la migration de tâches Le système d'exploitation voit les multicœurs comme les autres multiprocesseurs. Il assigne une tâche à chaque processeur. Sur les multiprocesseurs classiques, la localité des données ne peut être exploitée au maximum que si une tâche est toujours ordonnancée sur le même processeur. Le problème de migration des tâches a un impact moins important sur la performance dans les processeurs multicœurs . En effet, une grande partie de la hiérarchie de cache étant partagée, la migration d'une tâche d'un cœur sur un autre pose moins le problème de la migration des données dans les mêmes termes. Celle-ci est moins critique car on va chercher les données à une moindre profondeur dans la hiérarchie mémoire (généralement dans le cache L2 et non plus en mémoire). Les délais induits sont donc

moins importants.

Les processeurs multicœurs et le temps réel

Les processeurs multicœurs sont des multiprocesseurs, ils sont donc soumis aux même contraintes vis à vis du partage. Par exemple, la cohérence des caches doit être assurée.

Le partage qui est succesptible de nuire à la prévisibilité du temps d'exécution des tâches est celui des structures externes aux cœurs, en particulier les caches et surtout le réseau d'interconnexion. Tout ce qui entre et sort de tous les cœurs passe par ce réseau. C'est, le plus souvent, la structure partagée la plus sollicitée dans les processeurs multicœurs.

Dans [107], Rosen et al. décrivent une solution pour implanter des applications temps réel prévisibles sur des systèmes multicœurs. Les auteurs proposent d'utiliser un bus partagé selon une politique basée sur TDMA[32]. C'est une politique statique qui utilise des priorités prédéfinies. L'ordre d'allocation du bus aux cœurs est stocké dans une mémoire directement reliée à l'arbitre du bus. Cette technique requiert de connaître, a priori, l'ensemble des tâches à ordonnancer pour prévenir des contentions qui pourraient entrainer un allongement des latences mémoire.

Paolieri et al. proposent une architecture multicœurs conçue pour permettre l'analyse temporelle [92]. Ce processeur peut exécuter conjointement des tâches temps réel et des tâches sans contrainte temporelle. Le délai maximum que le partage des structures peut induire sur une tâche est borné. Un mode de calcul du pire temps d'exécution est supporté par le processeur. Dans ce mode, chaque tâche temps réel est temporellement isolée. Lors de chaque accès à une ressource partagée, ce délai maximum est ajouté artificiellement au temps d'exécution de la tâche. C'est une architecture à 4 cœurs. La mémoire et le cache L2 sont partagés entre les cœurs au travers d'un bus partagé. Les accès au bus sont arbitrés à deux niveaux : chaque cœur possède un arbitre qui gère les requètes vennant de ce cœur, et un autre, arbitre les requêtes entre les cœurs. Les tâches temps réel sont servies dans l'ordre des requêtes. La durée du traitement des requêtes étant constante, le délai maximum que chaque requête peut subir est connu. Les auteurs ont évalué leur architecture grâce à des méthodes dynamiques d'évaluation du pire temps d'exécution. L'utilisation de cette politique d'ordonnancement par tourniquet semblent pouvoir être utilisée avec des méthodes statiques.

[32]Time Divison Multiple Access

2.5 Conclusion

Les performances des processeurs modernes sont dues à l'exploitation par ceux-ci du parallélisme d'instructions et du parallélisme de tâches.

L'exploitation du parallélisme d'instructions est permise par la structure pipelinée des processeurs. L'exploitation de techniques comme l'exécution dynamique ou spéculative estompent les effets des facteurs limitants du parallélisme d'instructions que sont les dépendances structurelles, de données et de contrôle.

Le parallélisme de tâches est, quant à lui, rendu possible par la multiplication des processeurs, dans une même puce ou non, et par le partage d'un même cœur par plusieurs flots de contrôle. L'efficacité de cette méthode est directement liée aux politiques de partage des ressources internes du processeur, choisies lors de sa conception. En particulier, les points de sérialisation de l'exécution sont des facteurs limitant l'exploitation du parallélisme de tâche.

Le calcul du pire temps d'exécution de chaque tâche est essentiel pour l'ordonnancement des systèmes temps réel. Pour l'estimer, les méthodes dynamiques se basent sur l'exécution ou la simulation du code alors que les méthodes statiques se basent sur son analyse. Les systèmes temps réels stricts exigent que le pire temps d'exécution ne soit pas sous-estimé et demandent donc de privilégier les méthodes statiques. Dans ces dernières, la partie de l'analyse qui prend en compte l'architecture matérielle qui exécute les tâches est l'analyse de bas niveau. Elle nécessite une modélisation du processeur.

La précision du pire temps d'exécution évalué permet un meilleur dimentionnement des systèmes. Dans une analyse statique, il y a deux sources principales de surestimation : les informations de flots et la modélisation du processeur. Pour la première, elle est souvent due à l'imprécision dans l'évaluation des bornes de boucles. Pour la modélisation du processeur, c'est la modélisation des aspects dynamiques de l'exécution qui est difficile. En effet, il est difficile de connaitre l'état d'un processeur lorsqu'un bloc de base commence à s'exécuter ou encore de modéliser une politique d'ordonnancement lors d'une sérialisation dans un processeur SMT.

Le chapitre suivant propose une approche de ce problème en présentant un processeur SMT conçu pour assurer l'isolement temporel d'une tâche. Ce processeur, CarCore, est ensuite modélisé et une méthode d'analyse statique de la tâche temps réel en vue de l'estimation de son pire temps d'exécution est détaillée. Elle se base sur la technique d'énumération des chemins implicites. L'analyse de bas niveau est faite grâce aux graphes d'exécution.

Chapitre 3

Analyse de CarCore

3.1 Présentation de CarCore

CarCore [121] est un processeur multiflots simultanés conçu par l'équipe du processeur Ungerer de l'université d'Augsbourg (Allemagne).Nous avons choisi d'étudier ce processeur parce que les caractéristiques recherchées lors de sa conception permettent un isolement temporel des tâches qu'il exécute.

Le jeu d'instructions qu'il utilise est un sous-ensemble de celui du TriCore d'Infineon [59]. Il contient des instructions de 16 et 32 bits (on les reconnaît en regardant le bit 0 de l'instruction) mais est dépourvu des instructions DSP[33]. Il contient des instructions arithmétiques et logiques (travaillant sur des données de 8, 16, 32 et 64 bits), ainsi que des instructions de contrôle. Le jeu d'instructions du TriCore contient des instructions permettant de faire des opérations sur le pipeline, celles-ci ne sont pas implantées dans CarCore.

L'adressage des données peut se faire selon 9 modes. Le code opération est découpé en 2 à 4 parties suivant les instructions. Ces parties peuvent être réparties dans l'instruction selon 39 formats (14 formats pour les instructions 16 bits et 25 formats pour les instructions 32 bits). Les instructions (et les données en mémoire) sont codées selon le format petit-boutiste (little endian). Les adresses sont codées sur 32 bits.

CarCore est un processeur multiflots simultanés qui peut exécuter jusqu'à quatre tâches. Il est scalaire et exécute les instructions dans l'ordre du programme. Il est dépourvu de cache, de prédicteur de branchements et de mécanisme de renommage.

[33]Digital signal processing

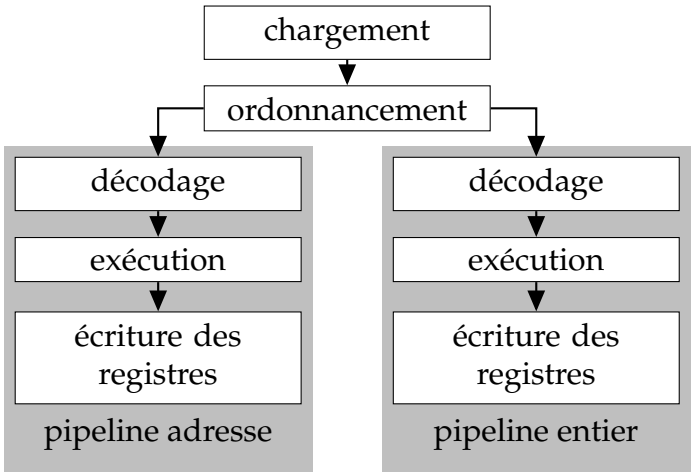


FIG. 3.1 – Le pipeline de CarCore

3.1.1 Le pipeline

Le pipeline de CarCore comporte 5 étages :
- chargement des instructions depuis la mémoire,
- ordonnancement : choix de la tâche à exécuter,
- décodage : identification de l'instruction lancée et des mécanismes de contrôle que son exécution nécessite,
- exécution,
- écriture dans les registres des résultats calculés dans l'étage précédent.

Le pipeline est dédoublé sur les trois derniers étages comme le montre la figure 3.1. Les instructions sont aiguillées vers l'un ou l'autre pipeline en fonction de leur code opération. Les instructions de chargement et de rangement mémoire ainsi que certains branchements vont dans le pipeline adresse, et les autres instructions vont dans le pipeline entier. CarCore ne dispose pas d'unité fonctionnelle pour les calculs sur les nombres à virgule flottante.

Le chargement

La figure 3.2 illustre le chargement des instructions. Le bus de données a une largeur de 64 bits. A chaque cycle, entre deux et quatre instructions peuvent être chargées. Les instructions chargées dans un même cycle appartiennent toujours à la même tâche.

Les instructions chargées sont placées dans une fenêtre d'instructions partagée statiquement entre les tâches actives. Chaque tâche dispose d'un espace de 48 octets, ce qui représente entre 12 et 24 instructions.

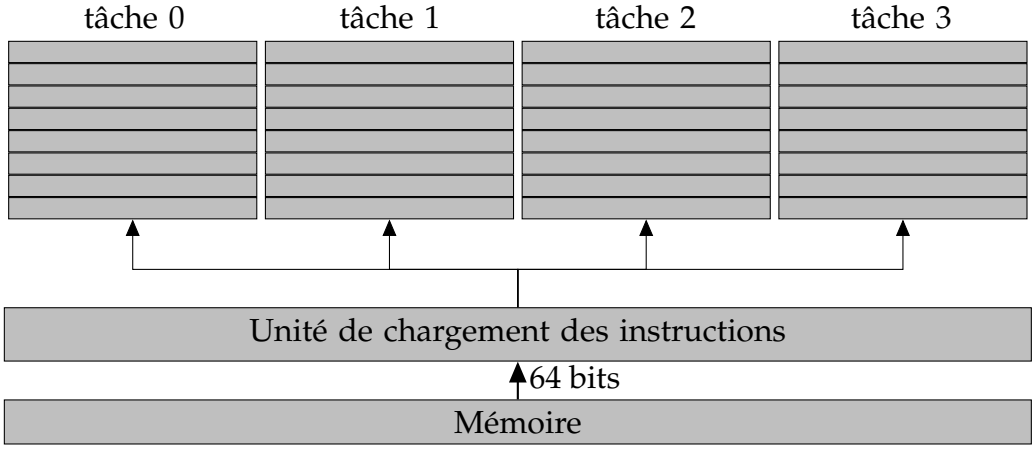


FIG. 3.2 – Le chargement des instructions

Afin d'éviter les débordements de la fenêtre d'instructions, une tâche dont la fenêtre est pleine aux deux tiers n'est pas éligible pour le chargement. A contrario, une tâche dont la fenêtre est aux deux tiers vide est préférée pour le chargement afin d'éviter les ruptures de débit. Si plusieurs tâches sont éligibles, le chargement se fait en accord avec la politique d'ordonnancement à priorité fixe (FPP[34]). Une priorité est associée à chaque tâche. C'est toujours la tâche de plus haute priorité qui est choisie pour le chargement. Si elle est bloquée, c'est à dire si elle n'a pas d'instruction qui peut être lancée, la tâche de priorité suivante est choisie et ainsi de suite.

L'ordonnancement des instructions

Les instructions sont ensuite retirées des fenêtres d'instructions et traitées dans le reste du pipeline. Le choix de la tâche à ordonnancer se fait par la politique d'ordonnancement FPP. A chaque tâche est associée une priorité et c'est toujours la tâche qui a la plus haute priorité qui est privilégiée. Si celle-ci est bloquée, c'est-à-dire si elle n'a pas d'instruction prête, c'est la tâche de priorité immédiatement inférieure qui est choisie et ainsi de suite.

Les instructions sont lancées dans le pipeline adéquat. Ce choix est possible avant le décodage car il suffit de regarder le bit 1 du mot mémoire de l'instruction pour déterminer dans quel pipeline elle doit être lancée.

Si une instruction de la tâche t_i est lancée dans le pipeline adresse, on doit attendre qu'elle soit sortie du pipeline avant de lancer une instruction de la même tâche. Les instructions qui s'exécutent dans l'autre pipeline ne souffrent pas de la même limita-

[34]Fixed Priority Preemptive

tion.

Le fonctionnement de cet étage est donc le suivant :

- d'abord, l'instruction suivante de la tâche la plus prioritaire est lancée dans le pipeline auquel elle est destinée. Si la tâche principale est bloquée, on recherche une instruction de la tâche de priorité inférieure.
- ensuite, si l'instruction a été lancée dans le pipeline adresse, on cherche une instrruction d'une tâche de priorité inférieure à lancer dans le pipeline entier. Si l'instruction a été lancée dans le pipeline entier, on cherche à lancer l'instruction suivante de la même tâche dans le pipeline adresse. Si on ne peut pas (il s'agit d'une istruction qui doit aussi être lancée dans le pipeline entier), on cherche parmi les instructions des tâches de priorité inférieure une instruction à lancer dans le pipeline adresse.

Le reste du pipeline

Dans les autres étages du pipeline, chaque instruction ne reste qu'un cycle. Les instructions ne sont jamais bloquées dans ces étages. Lorsqu'un blocage intervient, c'est toujours lors de l'ordonnancement.

Pour cela, les accès mémoires sont exécutés en deux étapes. La première étape correspond à l'envoi de l'opération mémoire à l'unité de gestion de la mémoire. Après cette étape, l'instruction poursuit son chemin dans le pipeline. La seconde étape correspond au résultat de la transaction mémoire. L'unité de gestion de la mémoire dispose d'un tampon pour chaque tâche, destiné à stocker les résultats des transactions mémoires en attente d'écriture dans les registres. Ce dispositif permet de ne pas bloquer l'ensemble des tâches activers lors d'un accès mémoire, mais seulement la tâche qui fait l'accès. Ainsi une tâche de priorité plus basse ne peut pas induire de délai pour une tâche de priorité plus élevée. L'attente de la donnée se fait toujours dans la fenêtre d'instructions avant que l'instruction ne soit lancée. Aucune instruction n'est bloquée après l'étage d'ordonnancement.

Certaines instructions comme les appels et retour de sous programmes, sont complexes et ne peuvent être exécutées en un cycle. Pour éviter les blocages du pipeline, elles sont microcodées. Les microinstructions sont exécutées comme des instructions normales à ceci près qu'elle n'ont pas besoin d'être chargées et qu'elles s'exécutent toujours dans le pipeline adresse. Pour le reste, elles sont traitées comme des instructions normales, en particulier, la priorité de la tâche dont elles sont issues leur est appliquée.

3.2 Modélisation de CarCore

Dans le processus de calcul de WCET que nous avons choisi d'utiliser, la prise en compte de l'influence du processeur se fait dans l'analyse temporelle et nous avons choisi d'utiliser les graphes d'exécution pour faire cette prise en compte afin que nos résultats soient utilisables dans le cadre du temps réel strict.

La technique des graphes d'exécution permet de prendre en compte la plupart des caractéristiques architecturales de CarCore. Certaines d'entre elles n'avaient pas été explorées avant ce travail, ce sont : le jeu d'instructions de taille variable, la politique d'ordonnancement, la politique de chargement qui évite les problèmes d'approvisionnement des fenêtres d'instructions, les pipelines multiples, les accès mémoires en deux phases, les instructions microcodées.

3.2.1 Les graphes d'exécution

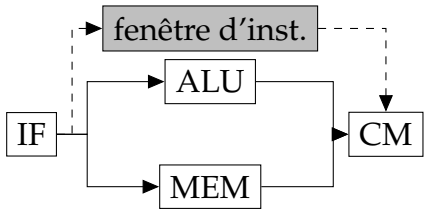
Un graphe d'exécution est un graphe orienté [73]. Il modélise l'exécution d'un bloc de base dans le processeur. Ses nœuds sont des couples (étage,instruction) et ses arcs représentent les dépendances entre les nœuds.

Construction d'un graphe d'exécution

Pour chaque instruction du bloc, on construit les nœuds correspondant à chaque étage du pipeline. Pour CarCore, par exemple, on construit 5 nœuds par instruction. Puis on crée les arcs de manière à représenter tous les liens de précédence possibles lors de l'exécution du bloc. Deux types de précédences peuvent survenir :

- le nœud A doit toujours être exécuté avant le nœud B. On construit alors un arc plein de A vers B. On s'en sert, par exemple, pour représenter la traversée du pipeline par les instructions.
- le nœud B peut être exécuté avant le nœud A mais il peut aussi l'être en même temps. Ce type de situation survient, par exemple, dans les processeurs superscalaires. Pour représenter ce type de précédence, on utilise un arc barré que nous avons introduit dans [6].

Pour illustrer la construction d'un graphe d'exécution et la manière dont il modélise les structures architecturales d'un processeur, nous allons utiliser l'exemple d'un bloc de base exécuté par un processeur très simple. Ce processeur est illustré par la figure 3.3(b). C'est un processeur à trois étages : chargement des instructions, exécution et enfin retrait. Le second étage comprend deux unités fonctionnelles, l'une pour



(b) Le processeur

I_0 : `r0 = MEM[@x]`
I_1 : `r1 = r0 + 8`
I_2 : `r2 = MEM[@y]`
I_3 : `r3 = r10 + 12`
I_4 : `r4 = r2 + r3`

(a) Le code

FIG. 3.3 – Un exemple de processeur et de code pour illustrer la construction des graphes d'exécution

les opérations arithmétiques et logiques, l'autre pour les opérations mémoire. Les instructions sont exécutées dans l'ordre du programme. Les étages de chargement et de retrait peuvent traiter deux instructions par cycle. Pour l'étage d'exécution, si deux instructions qui se suivent dans le programme utilisent des unités fonctionelles différentes (mémoire et ALU), elles peuvent être exécutées en même temps. La latence de chaque étage est d'un cycle et la fenêtre d'instructions peut contenir jusqu'à quatre instructions. Ce processeur ne comporte pas de cache ni de mécanisme de prédiction de branchement.

La figure 3.4 présente le graphe d'exécution associé au bloc de base. Pour chaque instruction, on a trois nœuds qui correspondent aux trois étages du pipeline.

Les arcs horizontaux modélisent la progression des instructions dans le pipeline alors que les arcs verticaux représentent l'exécution des instructions dans les étages. Ainsi, deux instructions successives peuvent être chargés successivement ou simultanément, l'arc barré entre $IF(I_0)$ et $IF(I_1)$, par exemple, représente cela. Mais les instructions ne peuvent être chargées que deux par deux, comme le montre l'arc plein entre $IF(I_0)$ et $IF(I_2)$, par exemple. I_2 ne peut pas être chargée dans le même cycle que I_0.

Les arcs barrés, entre $MEM(I_0)$ et $ALU(I1)$ par exemple, expriment que ces deux instructions peuvent être exécutées en même temps (elles utilisent des unités fonctionelles différentes).

Les arcs entre $CM(I_2)$ et $MEM(I_4)$, par exemple, représentent les dépendances de données. Içi, c'est une dépendance vraie :, la donnée contenue dans le registre `r2` produite par l'instruction I_2 est utilisée par I_4. Le processeur n'est pas muni de mécanisme d'envoi.

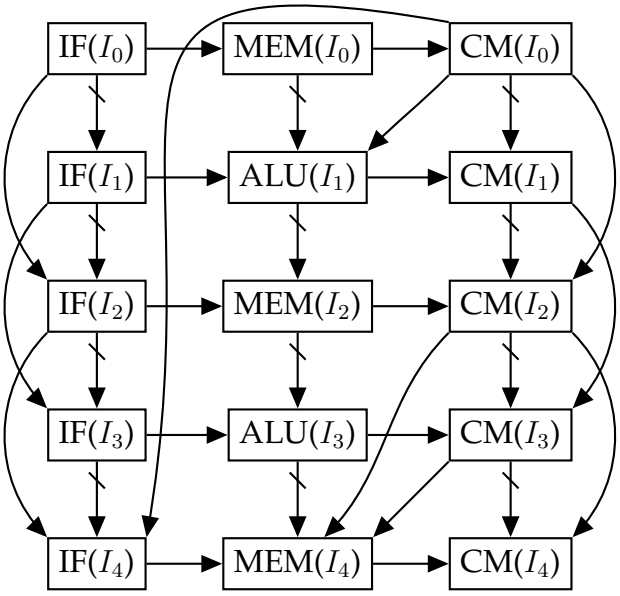


FIG. 3.4 – exemple de graphe d'exécution

L'arc plein entre $CM(I_0)$ et $IF(I_4)$ modélise la taille de la fenêtre d'instructions. I_4 ne peut pas être chargée avant que I_0 ne soit retirée du pipeline parce qu'il n'y a pas d'emplacement disponible dans la fenêtre d'instructions.

Détermination du pire temps d'exécution du bloc à partir de son graphe d'exécution

La méthode de calcul que nous utilisons est celle décrite dans [105]. Nous avons présenté la modélisation du contexte d'exécution dans la section 2.2.2, nous allons maintenant détailler le calcul de l'évaluation du pire temps d'exécution d'un bloc de base.

Calcul des dates auxquelles les nœuds sont prêts Les vecteurs $\mathcal{E}$ et $\mathcal{D}$ indiquent respectivement :

- le fait que la date à laquelle le nœud est prêt dépende ou non de chaque ressource du contexte d'exécution,
- le délai minimal entre la disponibilité de la ressource et la date à laquelle le nœud est prêt.

Les deux vecteurs sont d'abord initialisés. L'élément $\mathcal{E}_N^r$ est vrai si le nœud N utilise la ressource r et d_N^r est à 0, ce qui signifie que le nœud N ne peut commencer que lorsque la ressource r est disponible. Puis le graphe est parcouru dans l'ordre topolo-

Algorithme 3.1 Algorithme de propagation du calcul des vecteurs $\mathcal{E}$ et $\mathcal{D}$ dans un graphe d'exécution

```
for all predecesseur P du nœud N do
  if l'arc P → N est barré then
    lat = 0 ;
  else
    lat = l_P ;
  end if
  for all ressource r de R do
    if e_P^r == 1 then
      if e_N^r == 0 then
        e_N^r = 1 ;
        d_N^r = d_P^r + lat ;
      else
        if (d_P^r + lat) > d_N^r then
          d_N^r = d_P^r + lat ;
        end if
      end if
    end if
  end for
end for
```

gique et les dépendances sont propagées de nœud en nœud. Les délais sont aussi propagés selon l'algorithme 3.1. Le délai relatif à une ressource est le plus grand délai de ses prédécesseurs éventuellement majoré de la latence de la ressource. L'algorithme 3.1 détaille cette propagation. La latence du nœud P est notée l_P.

Calcul des coûts des blocs de base Après le calcul des dates auxquelles les nœuds sont prêts, les coûts des blocs de base peuvent être calculés en fonction du contexte qui est exprimé par le vecteur des dates de disponibilité des ressources. Si L_B est le dernier nœud du bloc, le temps d'exécution du bloc est donné par $t_B = \bar{\rho}_{L_B} + l_{L_B}$ où $\bar{\rho}_{L_B} = max_{r \in R}(\bar{e}^r_P.(\bar{d}^r_P + a^r))$ ou a^r est la date de disponnibilité de la ressource r. Pour calculer le coût du bloc (comme il est défini dans la section 2.1.2.0), il suffit que le bloc soit étendu par un préfixe d'au moins une instruction. On a alors

$$C_B = \bar{\rho}_{L_B} + l_{L_B} - \bar{\rho}_{L_P} - l_{L_P} \tag{3.1}$$

Contention pour une ressource

L'équation 3.1 s'écrit :

$$C_B = \Delta(L_B, L_P) + l_{L_B} - l_{L_B}$$

avec $\Delta(L_B, L_P) = \rho_{L_B} - \rho_{L_P}$.

Notons λ la ressource liée au dernier étage du pipeline. Tous les nœuds liés à cet étage dépendent de cette ressource car les instructions sont retirées du pipeline dans l'ordre du programme (le dernier étage est l'étage de retrait). Toutes les ressources de R sont libérées avant λ. On peut calculer le nombre de cycles avant lequel chaque ressource de R doit être libérée (avant λ) en considérant les étages ou elles sont utilisées. Notons α_r ce délai. On peut calculer une borne supérieure de $\Delta(L_B, L_P)$:

$$\Delta(L_B, L_P) \leq max_{r \in R}(\delta^r(L_B, L_P)) \tag{3.2}$$

avec $\delta^r(L_B, L_P) = e^r_{L_B}.(d^r_{L_B} - e^r_{L_P}.d^r_{L_P} - (1 - e^r_{L_P}).(d^\lambda_{L_P} + \alpha^r))$.

La preuve est donnée en annexe A.

3.2.2 Modélisation de CarCore par des graphes d'exécution

Le jeu d'instructions

Le jeu d'instructions utilisé par CarCore comporte des instructions de 16 bits et des instructions de 32 bits. Elles sont discriminées par le bit de rang 0 de l'instruction.

Le bus de données a une largeur de 64 bits. A chaque cycle, entre 2 et 4 instructions peuvent être chargées : 2 instructions de 32 bits, 4 instructions de 16 bits ou une instruction de 32 bits et 2 instructions de 16 bits. Dans le cas où une instruction de 32 bits suit une paire d'instructions de 32 et 16 bits, seules les deux premières peuvent être chargées.

Pour modéliser cela, nous utilisons les arcs qui séparent les nœuds de chargement des instructions. Lorsque les instructions peuvent être chargées dans le même cycle, leurs nœuds de chargement sont séparés par un arc barré. On utilise un arc plein pour deux instructions qui ne peuvent pas être chargées dans le même cycle.

La figure 3.5 montre la modélisation sur les quatre cas cités ci-dessus. Dans cette figure, les nœuds des instructions chargées dans le même cycle sont en gris. Les instructions seize bits en gris clair et les instructions trente deux bits en gris plus foncé.

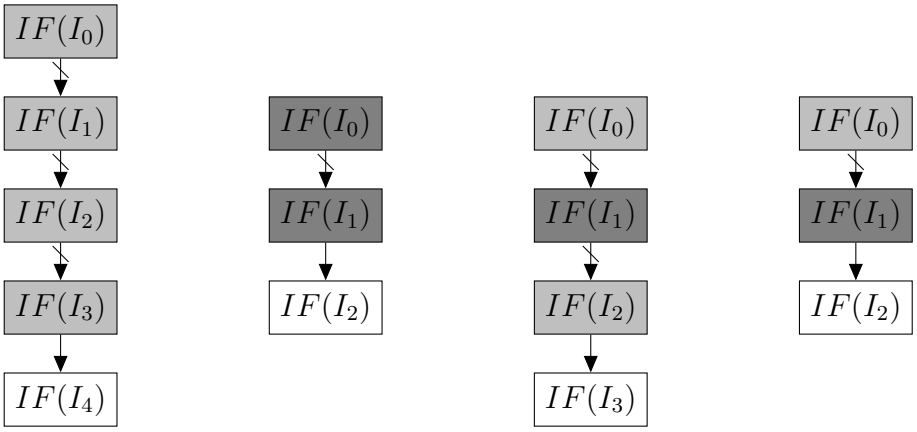


FIG. 3.5 – Modélisation du chargement des instructions de taille variable

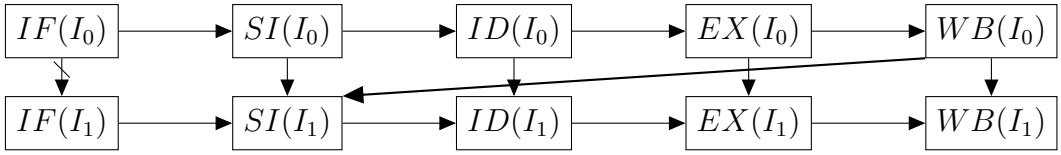


FIG. 3.6 – Une instruction pour le pipeline adresse suivie de n'importe quelle autre

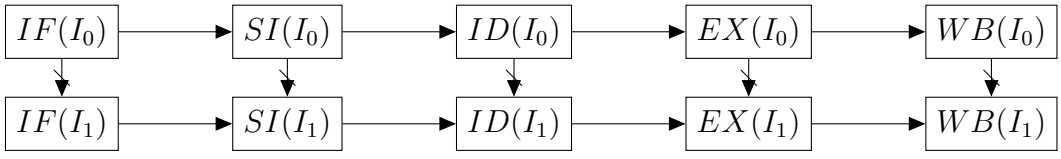


FIG. 3.7 – Une instruction pour le pipeline entier suivie d'une instruction pour le pipeline adresse

Les deux pipelines

CarCore est muni de deux pipelines, l'un que nous appellerons "adresse", l'autre que nous appellerons "entier". On discrimine les instructions qui vont dans l'un ou l'autre pipeline par le bit de rang 1 de l'instruction. Lorsqu'une instruction est lancée dans le pipeline adresse, l'instruction suivante de la même tâche doit attendre que celle qui la précède sorte du pipeline pour être lancée. Cette limitation ne s'applique pas au pipeline entier, on peut ainsi lancer une instruction dans le pipeline adresse dans le même cycle que l'instruction qui la précède si celle-ci est destinée au pipeline entier.

Pour modéliser le blocage de la tâche, on utilise un arc plein entre le nœud d'écriture des registres de l'instruction destinée au pipeline adresse et le nœud d'ordonnancement de celle qui la suit, comme représenté sur la figure 3.6.

On utilise des arcs barrés entre les nœuds d'ordonnancement, de décodage, d'exécution et d'écriture des registres d'une instruction destinée au pipeline entier et l'instruction destinée au pipeline adresse qui la suit, comme sur la figure 3.7. Les arcs verticaux barrés entre les nœuds $SI(I_0)$ et $SI(I_1)$, $ID(I_0)$ et $ID(I_1)$, $EX(I_0)$ et $EX(I_1)$ et enfin $WB(I_0)$ et $WB(I_1)$ indiquent que les deux instructions peutvent parcourir leur pipeline respectif en même temps (ou dans l'ordre). Mais I_1 ne peut pas être lancée dans son pipeline si I_0 ne l'est pas.

Les accès mémoire

Les accès mémoire se font en deux phases. D'abord la requête est envoyée et l'instruction continue son chemin dans le pipeline. Ensuite l'accès proprement dit est réalisé.

Pour un rangement en mémoire, cette seconde phase n'a pas d'impact. La donnée à écrire est transmise à l'unité de gestion de la mémoire lors de la requête. Il n'est pas nécessaire d'attendre que la transaction soit terminée pour continuer l'exécution du programme.

Pour une lecture en mémoire, l'instruction qui a besoin de la donnée l'attend dans la fenêtre d'instructions. On doit donc mettre un arc plein entre le nœlud d'exécution de la lecture et le nœud d'ordonnancement de l'instruction qui attend la donnée. Or toutes les instructions mémoire sont exécutées dans le pipeline adresse, et un arc plein indique déjà le blocage de la tâche (comme dans la figure3.6). De plus, la latence de la mémoire de données est de trois cycles, ce qui est inférieur ou égal au temps nécessaire à une instruction pour traverser le pipeline. La requête est lancée dans l'étage d'exécution.

La combinaison de ces deux caractéristiques permet de se passer de l'arc entre la lecture et l'instruction qui attend la donnée.

Les instructions microcodées

Les instructions à latence longue sont exécutées sous la forme de microinstructions. Celles-ci sont identiques aux autres instructions mais elles n'ont pas besoin d'être chargées et s'exécutent toujours dans le pipeline adresse, ce qui assure que les autres instructions attendent qu'elles soient terminées.

Parmi les microinstructions, certaines font des accès mémoires. Pour modéliser une instruction microcodée I, nous avons choisi d'utiliser la latence L_I^{SI} de son étage d'ordonnancement. Nous l'exprimons comme la somme des latences (l_i) des microinstructions i qui composent I.

$$L_I^{SI} = \sum_{i \in I} l_i$$

Les latences (l_i) sont un peu particulières puisqu'elle correspondent aux latences des microinstructions i pour l'étage SI (l_i^{SI}) auxquelles on ajoute les latences liées aux éventuels accès mémoires (l_i^{MEM}).

$$\forall i \in I, l_i = l_i^{SI} + l_i^{MEM}$$

Cette modélisation est correcte parce que les microinstructions sont toujours exécutées par le pipeline adresse, elles obéissent donc à la règle que veut que d'autres instructions de la même tâche ne peuvent pas être exécutées avant que l'instruction microcodée ne soit terminée. Les seules interactions qui peuvent se produire avec d'autres instructions se font au travers de la première (arcs entrant de l'instruction microcodée) et de la dernière (arcs sortant de l'instruction microcodée) microinstruction. L'instruction microcodée se comporte comme une instruction normale à latence longue.

3.3 Implantation de la modélisation

En plus de la modélisation théorique, un gros travail d'implantation a été fait afin de produire des résultats expérimentaux. Pour cela, nous avons utilisé des outils existants : GLISS et OTAWA. Le premier a permis de générer des bibliothèques de fonctions qui sont utilisées par le second pour décoder et émuler les binaires compilés pour le TriCore d'Infineon. OTAWA fournit un ensemble d'utilitaires qui permettent le calcul du pire temps d'exécution de tâches. Nous les avons adapté pour qu'ils prennent en

compte notre modèle. Une difficulté a été que les deux outils sont actuellement en développement. Nous avons donc dû nous adapter à leur constante évolution. Cela a aussi été un avantage car nous avons pu suggérer des ajouts de fonctionnalité dont nous avions besoin.

3.3.1 Documentation sur CarCore

Pour ce qui est de la prise en charge du jeu d'instructions, l'obtention des informations n'a pas été difficile, CarCore utilise un sous-ensemble du jeu d'instructions du TriCore d'Infineon, lequel est documenté dans le tome 2 du manuel d'utilisation du TriCore 1 [59]. La version que nous avons utilisée est la version 1.3.6 qui a servi à l'équipe de l'université d'Augsbourg lors de la réalisation du simulateur. Pour déterminer le sous-ensemble du jeu d'instructions du TriCore utilisé par CarCore, nous avons analysé le simulateur fourni par l'équipe de l'université d'Augsbourg.

Pour l'analyse de la microarchitecture, nous avons passé trois mois à l'université d'Augsbourg, en compagnie de l'équipe qui a conçu CarCore. Nous avons donc pu obtenir les informations directement des concepteurs. Dans une autre phase, nous avons analysé le code source de leur simulateur pour connaître certains détails d'implantation.

3.3.2 GLISS

GLISS est un outil qui permet de générer un simulateur fonctionnel (qui interprète les instructions) à partir d'une description du jeu d'instructions écrite en nML. Dans le simulateur généré, il y a une bibliothèque de fonctions qui permettent, entre autres, de charger et de décoder chaque instruction. OTAWA utilise ces fonctions pour interpréter les binaires compilés pour le TriCore. Pour ce faire, nous avons aussi développé un module de chargement pour OTAWA, que nous présenterons dans la section suivante.

Vue générale

GLISS est basé sur l'outil Sim-nML développé par l'Indian Institute of Kanpur. La production de l'émulateur d'un jeu d'instructions se fait à partir de sa description en langage nML. Ceci produit un émulateur en code C qui doit être compilé grâce à gcc pour générer une bibliothèque qui peut ensuite être liée à un simulateur. La figure 3.8 donne une vue d'ensemble du procédé. GEP est un élément de GLISS qui permet de générer les fonctions d'émulation qui seront ensuite compilées par gcc sous forme d'une

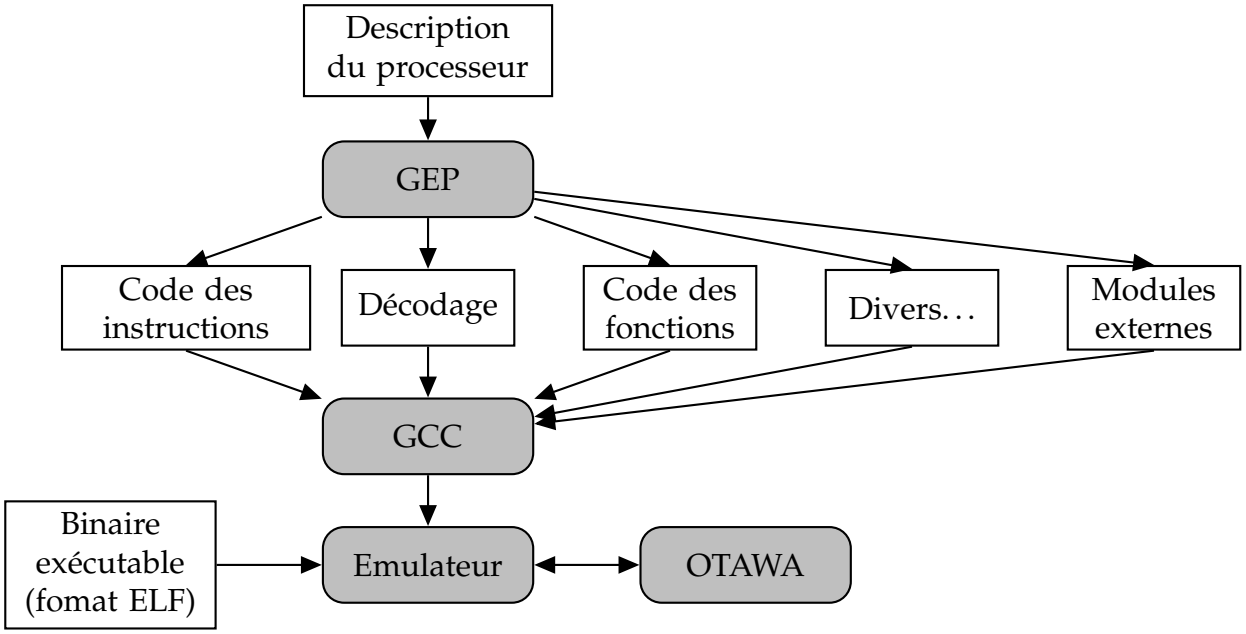


FIG. 3.8 – vue d'ensemble de la production d'un emulateur par GLISS

bibliothèque.

Le lien entre l'émulateur généré et OTAWA se fait au travers d'un petit nombre de fonctions et de structures de données. Le tableau 3.1 présente un résumé des fonctions que nous avons utilisées et de leur rôle.

Le langage nML

Le langage nML permet de décrire le jeu d'instructions utilisé. La description comporte les ressources (mémoire, registres), les modes d'adressage et les instructions.

Nous allons prendre des exemples dans le jeu d'instructions du TriCore d'Infineon pour illustrer ce que nous avons fait.

La première chose à faire est de décrire les caractéristiques des ressources du processeur comme le boutisme (endianness), la taille du bus d'adresse, l'ensemble des registres du processeur et la mémoire :

Le code de la figure 3.9 montre certaines de ces déclarations. Les commentaires sont introduits par `//`. Les variables sont définies à l'aide du mot clef `let`. Ainsi, la variable `MEM_SIZE` est utilisée pour définir la taille en bits du bus d'adresse.

Des types sont ensuite déclarés. Ils sont ensuite utilisés pour typer les registres et les paramètres des instructions. Leur déclaration commence par le mot clef `type` et le mot clef `card` permet de spécifier leur taille. Ainsi, le type word est composé de 32

nom	description
`iss_fetch`	Cette fonction récupère l'instruction, sous forme d'une suite d'octets, à partir d'une adresse passée en paramètre.
`iss_decode`	Cette fonction reçoit une suite d'octets correspondant à une instruction. Elle renvoie une instance de l'instruction décodée. Cette instance est une structure qui contient plusieurs champs dont l'un nommé `ident` contient l'identité de l'instruction, c'est à dire une valeur symbolisant son code opération et un tableau `instrinput` qui contient tous les paramètres de l'instruction comme définis dans la description nML.
`iss_free`	Cette fonction libère la mémoire allouée pour une instruction lors de l'appel à `iss_decode`.
`iss_disasm`	Cette fonction prend en paramètres une instruction et un tampon qu'elle remplie avec l'instruction désassemblée en utilisant le champs `syntax` de la description de l'instruction.

TAB. 3.1 – Résumé de l'interface de l'émulateur généré par GLISS

```
let MEM_SIZE = 32
let M_is_little = 0 //the memory is little endian
let AREGS = 4
let DREGS = 4
let EREGS = 2

type index = card(AREGS)
type word = card(32)
type double_word = card(64)
type byte = card(8)
type index64 = card(EREGS)

reg A[2**AREGS,word] //adress registers
reg D[2**DREGS,word] //data registers
reg E[2**EREGS,double_word] //64 bits registers

mem M[2**32,byte]
```

FIG. 3.9 – exemple de déclaration des ressources

```
mode reg_a (r:index) = A[r]
   syntax = format("a%d",r)
   image = format("%4b",r)

mode reg_d (r:index) = D[r]
   syntax = format("d%d",r)
   image = format("%4b",r)


mode reg_e (r:index) = E[r]
   syntax = format("e%d",r)
   image = format("%4b",r)
```

FIG. 3.10 – exemple de déclaration des mode d'adressage

bits.

Les bancs de registres sont introduits par le mot clef `reg`, et on indique l'identificateur du banc et, entre crochets, le nombre et le type des registres. Par exemple, le banc de registres `A` contient 16 (2 puissance 4) registres de 32 bits.

On peut également définir des modes d'adressage. Comme le montre la figure 3.10, c'est le mot clef `mode` qui permet de déclarer les modes d'adressage. Chaque mode a une syntaxe et une image respectivement introduites par les mots clefs `syntax` et `image`. La syntaxe est la représentation du mode d'adressage dans le code assembleur. L'image représente la manière dont est codé le mode d'adressage dans le mot mémoire de l'instruction. Par exemple, le mode `reg_a` désigne un registre du banc `A` et comprend un paramètre de type index. Un registre est représenté en assembleur par a_n ou n est le numéro du registre. Et l'image du registre dans le mot mémoire d'une instruction comporte 4 bits qui représentent le numéro du registre.

Une instruction est définie à l'aide du mot clef `op`, accompagné de 3 attributs : `syntax`, `image` et `action`. Comme pour le mode d'adressage, la syntaxe correspond à la représentation de l'instruction dans le langage d'assemblage. L'image représente le mot mémoire qui code l'instruction dans le programme. L'action modélise la sémantique de l'instruction, qui sera reprise lors de la génération d'un simulateur fonctionnel.

La figure 3.11 donne un exemple pour l'instruction `mov` qui copie la valeur d'un registre dans un autre. Elle prend en paramètres deux registres. Son image est plus com-

```
op mov_reg (c:reg_d,foo1:card(4),foo2:card(4),b:reg_d)
  syntax = format("mov %s,%s",c.syntax,b.syntax)
  image = format("%s00011111%4b%s%4b00001011",
    c.image,foo1,b.image,foo2)
  action = { c = b; }
```

FIG. 3.11 – Exemple de déclaration d'une instruction

```
op j (const:card(32),d1:card(16),d2:card(8))
 predecode = {
  const = 0;
  const<16..23> = d2<0..7>;
  const<0..15> = d1<0..15>;
  if (const<23..23>==1) then
   const<24..31>=0xff;
  endif;
  const = 2*const;
 }
 syntax = format("j %d",const)
 image = format("%16b%8b00011101%0b",d1,d2,const)
 action = { PC = PC-4+const; }
```

FIG. 3.12 – exemple d'utilisation du pré-décodage

pliquée : les valeurs `0` et `1` contenues dans l'image correspondent au code opération de l'instruction. Comme on peut le voir, il est en plusieurs parties (il existe 39 formats d'instructions différents dans le jeu d'instructions du TriCore). De plus, nous avons dû ajouter deux paramètres `foo1` et `foo2` de 4 bits chacun (`%4b`). Ils correspondent aux deux parties de l'instruction qui ne sont pas codantes, c'est-à-dire que la valeur de ces bits dans l'instruction n'a pas de signification. Ils peuvent être indifféremment 0 ou 1 sans que l'instruction n'en soit affectée.

Une autre caractéristique du jeu d'instruction du TriCore qui a posé des problèmes lorsque nous l'avons décrit, est le fait que le codage des opérandes de certaines instructions est aussi découpé en plusieurs parties. Pour résoudre ce problème, nous avons utilisé l'attribut de pré-décodage supporté par GLISS. Il s'agit d'ajouter un faux paramètre à l'instruction et de l'utiliser pour reconstruire l'opérande.

La figure 3.12 donne l'exemple de l'instruction `j` qui est un branchement inconditionnel. Le faux paramètre est nommé `const`. Il correspond au décalage qu'on doit

```
op madd_u_reg(c: reg_e, d: reg_e, a: reg_d, b: reg_d)
 syntax  = format("madd.u %s, %s, %s, %s",
  c.syntax, d.syntax, a.syntax, b.syntax)
 image   = format("%s%s01101000%s%s00000011",
  c.image, d.image, b.image, a.image)
 action  = {
  result64 = d + coerce(card(32), a) * coerce(card(32), b);
  c = result64;
 }
```

FIG. 3.13 – exemple d'utilisation de la fonction "coerce"

appliquer au compteur de programme pour obtenir l'adresse de la prochaine instruction à charger. Son image dans l'instruction est de 0 bits. Les deux parties de l'opérande sont `d1` et `d2`, qui correspondent respectivement aux bits 0 à 15 et 16 à 23 de l'opérande (qui est représenté au total sur 32 bits). Les bits manquants sont reconstruits par une extension de signe (ils sont tous identiques au bit 23). Enfin, un bit est économisé dans le codage de l'instruction car toutes les adresses sont alignées sur 16 bits. Il faut donc décaler tous les bits d'un rang vers la gauche pour avoir la bonne valeur pour const. C'est le rôle de la multiplication par 2.

C'est la section `predecode` de la description de l'instruction qui se charge de cette reconstruction. Elle est exécutée avant le décodage, ce qui permet d'avoir la bonne valeur de `const` lorsque celle-ci doit être utilisée.

Comme on peut le voir dans l'exemple précédent, on peut utiliser, dans la section `predecode`, un langage qui comporte des instructions conditionnelles (if) et des opérations sur les champs de bits. Les structures itératives (boucles) sont aussi utilisables. On peut aussi utiliser ce langage dans l'attribut `action`. Enfin, il faut noter que ce langage fourni aussi une fonction `coerce` qui permet le transtypage. Son emploi est rendu indispensable par le fait que les opérations sont réalisées en langage C, qui prend en compte le type des opérandes avant de réaliser l'opération. Par exemple, si on veut réaliser une addition entre deux opérandes de 32 bits, pour avoir un résultat sur 64 bits, une conversion explicite des opérandes est indispensable avant de réaliser l'opération. L'utilisation de cette fonction est illustrée par la figure 3.13. L'instruction `madd` réalise la multiplication de deux registres de 32 bits (non signés dans le cas de `madd.u`) et ajoute le résultat à la valeur contenue dans un registre 64 bits et range la somme dans un registre 64 bits. Dans CarCore comme dans le TriCore, les registres 64 bits sont en réalité, constitués de deux registres 32 bits contigus.

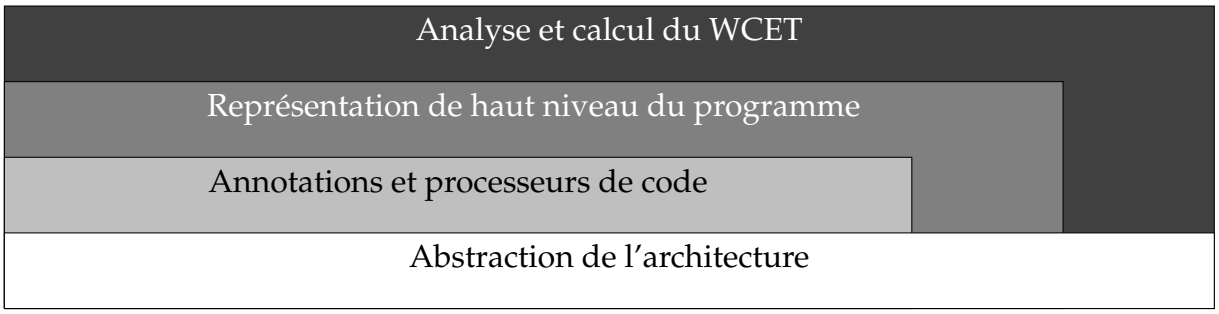

Analyse et calcul du WCET
Représentation de haut niveau du programme
Annotations et processeurs de code
Abstraction de l'architecture

FIG. 3.14 – L'empilement de couches du système

3.3.3 OTAWA

OTAWA[35] [20] est un logiciel développé dans l'équipe TRACES[36] de l'IRIT[37]. Il permet de calculer le pire temps d'exécution des programmes. Il présente une architecture ouverte et extensible dont l'objectif est d'implanter les outils présents et futurs d'analyse statique du pire temps d'exécution de tâches. Il est basé sur un empilement d'abstractions auxquelles sont associées des annotations qui permettent de stocker des informations. Le calcul du pire temps d'exécution est vu comme une chaîne d'analyses successives qui utilisent et produisent des annotations.

Présentation générale

La figure 3.14 montre la structure en couches du système. On peut remarquer que toutes les couches ont accès à l'abstraction de l'architecture. Celle-ci est la colonne vertébrale du système. Toutes les informations produites sont liées au code du programme traité.

La couche la plus basse fournit une abstraction de l'architecture matérielle aux couches supérieures. Comme le montre la figure 3.15, le chargeur a pour fonction de lire le programme binaire, les informations de flots et la configuration du matériel utilisé, et d'en tirer les informations nécessaires aux traitements ultérieurs. Celles-ci sont stockées dans la représentation du processus.

La seconde couche fournit des annotations qui peuvent être attachées aux éléments architecturaux de la première. Ces annotations sont aisément modifiables. Une annotation est une information typée repérée par un identifieur. Elles peuvent être associées à

[35]Open Tool for Adaptative WCET Analysis
[36]Traces stands for Research group on Architecture and Compilation for Embedded Systems
[37]Institut de Recherche en Informatique de Toulouse

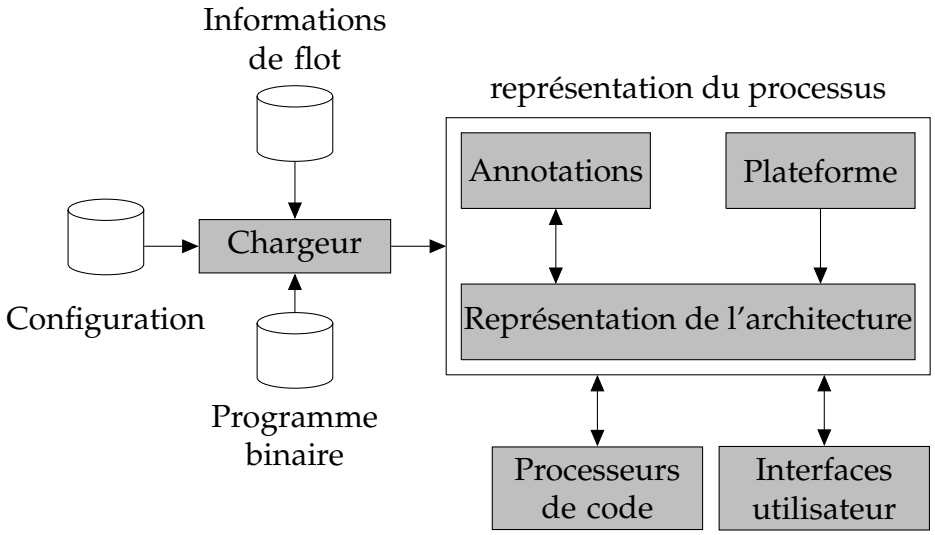


FIG. 3.15 – La couche d'abstraction de l'architecture

n'importe quel élément qui les supporte, comme l'abstraction de l'architecture ou une représentation de haut niveau du programme.

Ce système convient bien au calcul de pire temps d'exécution par sa généricité : chaque analyse apporte davantage d'informations qui sont capitalisées via les annotations. Ils fonctionnent en parcourant la représentation de l'architecture et en relevant les annotations fournies par les analyseurs précédents. Ils stockent leurs résultats dans de nouvelles annotations ou en modifiant des annotations existantes.

La combinaison des processeurs de code utilisées est souple : on peut les choisir en fonction de la précision et du temps de calcul désiré. Le système d'annotations rend aussi l'analyse non linéaire en rendant les analyseurs indépendants les uns des autres.

La couche "représentation de haut niveau du programme", est construite sur le système d'annotations. OTAWA supporte plusieurs représentations de haut niveau des programmes : le graphe de flot de contrôle, l'arbre syntaxique et l'arbre de contexte. Ils sont construits par des processeurs de code dédiés et liés à la représentation de l'architecture par des annotations. L'extensibilité permise par le système d'annotations autorise l'ajout ultérieur d'autres représentations.

OTAWA permet aussi de travailler sur des graphes de flots de contrôle virtuels. Ceux-ci sont des copies du graphe réel mais ils sont modifiables par les processeurs de code qui en ont besoin. Par exemple, une processeur de code qui analyse l'impact de la mémoire cache peut travailler sur un graphe de flot de contrôle virtuel et y découper des blocs de base pour les faire correspondre aux blocs de cache (l-blocs).

La couche la plus haute fournit des outils d'analyse et de calcul du pire temps d'exé-

cution. Ces derniers utilisent les informations données dans les autres couches. C'est la partie qui permet de faire les analyses nécessaires à l'estimation du pire temps d'exécution. Elle s'appuie sur les couches précédentes et bénéficie des fonctionnalités qu'elles fournissent.

Le système n'est pas lié à une méthode particulière de calcul de pire temps d'exécution. OTAWA fournit actuellement un ensemble d'outils nécessaires pour employer les méthodes par énumération des chemins implicites et par schéma temporel étendu (ETS[38]) [63]. Mais on peut envisager l'implantation d'autres méthodes d'analyse statique.

Le chargeur

Comme nous l'avons vu dans la section précédente, le rôle du chargeur est de permettre à OTAWA de lire les binaires compilés pour le TriCore. Pour cela, il utilise la bibliothèque de fonction produite par GLISS. Nous avons implanté les fonctionnalités nécessaires au fonctionnement des graphes d'exécution dans OTAWA, c'est-à-dire : reconnaissance de toutes les instructions, catégorisation des instructions de branchements et des instructions mémoires, détermination des adresses des branchements.

Ces informations sont ensuite utilisées dans OTAWA pour construire le graphe de flot de contrôle (branchement, appels et retours de sous-programmes, branchements conditionnels et leurs adresses cibles) ainsi que pour construire les graphes d'exécution comme nous le verrons dans la section suivante.

Pour la reconnaissance des instructions, la majeure partie du travail est assurée par GLISS. Dans le chargeur, il faut déclarer les bancs de registres et créer une instruction OTAWA pour chaque code opération.

La catégorisation des instructions consiste à spécifier si une instruction est un chargement ou un rangement en mémoire, une opération arithmétique, une instruction conditionnelle, un appel ou un retour de sous-programme. Pour ce faire, il faut d'abord charger et décoder l'instruction. Cela se fait en appelant les fonctions `iss_fetch`, `iss_decode` et `iss_disasm` de la bibliothèque générée par GLISS.

Pour les instructions de contrôle (branchements, appels et retour de sous-programmes), GLISS fournit le déplacement qui correspond à la cible du branchement.

La répartition des instructions que nous avons codées dans le chargeur pour TriCore/CarCore est résumée dans le tableau 3.2. La proportion est calculée par rapport

[38] Extended Timing Schema

Type d'instruction	Nombre	Proportion
Instructions conditionnelles	39	17,57%
Instructions de chargement	28	12,61%
Instructions de rangement	25	11,26%
Instructions de appel de sous-programme	2	0,91%
Instructions de retour de sous-programmes	2	0,91%
Instructions de branchement	37	16,67%
Instructions de branchement inconditionnels	3	1,35%
Total	222	100%

TAB. 3.2 – Les instructions codées dans le chargeur pour TriCore/CarCore

au nombre total d'instructions codées (dernière ligne du tableau). On remarque que la majorité des instructions sont des instructions arithmétiques. Il y a 23,87% d'instructions mémoires et 18,47% d'instructions de contrôle. On peut aussi préciser que la plupart des opérations existent en version 16 bits et 32 bits et que le jeu d'instructions comporte 7 modes d'adressage.

Le jeu complet d'instructions du tricore comprte 329 instructions dont certaines existent dans les 7 modes d'adressages. On a donc jusqu'à 2300 code opérations possibles au total. En fait il y en a moins car toutes les instructions n'existent pas dans tous les modes d'adressages. Nous en avons codé 222 sur ce total.

La construction des graphes d'exécution

Dans OTAWA, les graphes d'exécution sont construits à partir du graphe de flot de contrôle, des instructions créées par le chargeur et de la description du processeur faite en XML. Un graphe est généré pour chaque bloc de base et pour chaque contexte. Cependant, la méthode qui construit les graphes a dû être adaptée pour que les graphes produits correspondent à notre modélisation.

Le fichier qui décrit le processeur est reproduit dans la figure 3.16. La description comporte deux parties : les étages (`stages`) et les files et tampons (`queues`). Chaque étage reçoit :

- un identifiant (`ID`) qui permet à OTAWA de l'identifier de manière unique,
- un nom (`name`) qui sert au programmeur,
- une taille (`size`) qui correspond à la largeur de l'étage, c'est à dire à son degré de parallélisme,
- un type (`type`), il en existe 6 mais nous n'en utilisons que 2. Les autres sont utiles si on souhaite faire de la simulation structurelle. La valeur `FETCH` permet l'ob-

```
 <?xml version="1.0" encoding="UTF-8"?>
<processor class="otawa::hard::Processor">
<arch>op</arch> <model>CarCore</model><builder>OTAWA</builder>
        <stages>
                <stage id="IF">
                        <name>IF</name>
                        <width>4</width>
                        <type>FETCH</type>
                        <latency>1</latency>
                </stage>
                 <stage id="SI">
                        <name>SI</name>
                        <width>2</width>
                        <type>LAZY</type>
                        <latency>1</latency>
                </stage>
                 <stage id="ID">
                        <name>ID</name>
                        <width>2</width>
                        <type>LAZY</type>
                </stage>
                <stage id="EX">
                        <name>EX</name>
                        <width>2</width>
                        <type>LAZY</type>
                        <latency>1</latency>
                </stage>
                <stage id="WB">
                        <name>WB</name>
                        <width>2</width>
                        <type>LAZY</type>
                        <latency>1</latency>
                </stage>
        </stages> <queues>
                <queue>
                        <name>FETCH_QUEUE</name>
                        <size>4</size>
                        <input ref="IF"/>
                        <output ref="SI"/>
                </queue>
        </queues>
</processor>
```

FIG. 3.16 – Description du processeur CarCore en XML

tention des instructions depuis la mémoire et `LAZY` transmet simplement l'instruction de l'élément précédent à l'élément suivant,

– une latence (`latency`) qui est la latence de l'étage.

Dans notre description, l'étage de chargement à une largeur de 4 instructions puisque jusqu'à 4 instructions 16 bits peuvent être chargées dans le même cycle et les tous derniers étages ont une largeur de 2 pour représenter les deux pipelines de CarCore.

Les queues ont un nom, une taille qui correspond au nombre d'instructions qu'elles peuvent accueillir ainsi qu'une entrée (`input`) et une sortie (`output`) qui définissent leur position dans le pipeline. Ainsi, dans notre description, la fenêtre d'instructions se trouve entre les étages de chargement et d'ordonnancement et elle peut accueillir 8 instructions.

Le langage est plus puissant que ne le montre notre exemple. Il permet en particulier de définir des unités fonctionnelles pour chaque étage et une section de répartition (`dispatch`) permet d'aiguiller automatiquement les instructions vers l'une ou l'autre des unités fonctionnelles en fonction des catégories auxquelles elles appartiennent (attribut `kind` de l'instruction).

Pour que les graphes d'exécution produits soient conformes à notre modèle de CarCore, nous avons modifié la méthode qui les construit en intégrant la gestion des instructions de taille variable, l'aiguillage des instructions vers le bon pipeline, l'ajout des arcs qui modélisent le blocage du pipeline adresse lors de l'exécution des branchements et des chargements mémoires, le parallélisme d'exécution des instructions lorsqu'une instruction du pipeline adresse suit une instruction du pipeline entier, les latences des instructions mémoire, les instructions microcodées.

Pour la prise en compte des instructions de taille variable, nous avons supposé que le début d'une séquence de blocs n'est pas à cheval sur une fenêtre de chargement, c'est à dire que lors du chargement des premières instructions du bloc, seules des instructions de ce bloc sont chargées, ce qui correspond à ce qui est fait par le simulateur de CarCore. De plus, il faut tenir compte du fait que lorsqu'on rencontre un branchement pris, les instructions de la cible ne peuvent pas être chargées dans le même cycle que les instructions du bloc source.

Pour les autres fonctionnalités, il suffit de connaître la catégorie de l'instruction, et de faire la modification correspondante, soit ajouter un arc, soit modifier la latence.

La prise en compte des instructions microcodées a été plus problématique. Nous avons finalement choisi de les modéliser en modifiant la latence du nœud correspondant à leur étage d'ordonnancement Mais chaque instruction micro-codée se décompose en une séquence de micro-instructions qui lui est propre et dont la taille et donc

la latence, est variable d'une instruction à l'autre. Or OTAWA ne permet pas de remonter au code opération de l'instruction une fois celle-ci décodée. Nous avons utilisé une astuce qui a consisté à réserver une catégorie non utilisée.

3.4 Conclusion

CarCore est un processeur à exécution simultanée de flots qui assure l'isolement temporel de la tâche de plus haute priorité. L'analyse fine du simulateur de CarCore a été une première difficulté. Nous avons dû analyser le code source pour comprendre les détails de son fonctionnement. La version qui nous a été donnée par l'université d'Augsbourg est une version de développement.

Nous avons modélisé l'exécution de programmes par ce processeur en vue de calculer leur pire temps d'exécution par la méthode IPET. Cette modélisation se fait grâce à des graphes d'exécution.

Nous avons introduit la modélisation de la précédence partielle de l'exécution en ajoutant des arcs barrés au modèle initial. Nous avons ainsi pu modéliser l'exécution d'instructions de taille différente (16 et 32 bits), la présence de deux pipelines et la politique d'ordonnancement (priorité fixe). Nous avons encore modélisé les accès mémoires faits en deux phases. Nous avons enfin modélisé les instructions microcodées en utilisant la latence de leur ordonnancement.

L'implantation a été réalisé grâce à GLISS et OTAWA. Le premier nous a permis de générer une bibliothèque de fonctions qui prend en charge le décodage du jeu d'instructions du TriCore, utilisé par CarCore. Cet outil ne prenait pas en compte les instructions de taille variable. Nous avons aussi rencontré des problèmes avec le décodage du code opération et les constantes qui sont en plusieurs parties dans ce jeu d'instructions.

OTAWA nous a fourni les outils nécessaires au calcul du pire temps d'exécution comme la construction du graphe de flot de contrôle ou la résolution de systèmes d'équations entières. Nous avons réalisé un chargeur qui utilise les fonctions générées par GLISS pour exprimer les instructions dans le format utilisé par OTAWA. Cela permet en particulier la reconnaissance des instructions de contrôle et le calcul des adresses cibles des branchements qui sont nécessaires pour la construction du graphe de flot de contrôle. Nous avons aussi modifié le module de construction des graphes d'exécution afin de prendre en compte notre modélisation. Nous avons rencontré, avec OTAWA, des problèmes mineurs, liés au fait que c'est un logiciel en cours de développement.

Chapitre 4

Performances pire cas de l'architecture CarCore

4.1 Méthodologie

Pour évaluer la précision de notre modèle, nous avons comparé les temps d'exécution des blocs de base obtenus par simulation et les pires temps estimés avec OTAWA. Pour ce faire, nous avons choisi de faire nos mesures sur un petit nombre de benchmarks courramment utilisés dans la communauté du WCET. Cette section traite du choix de ces programmes et de leur préparation en vue de leur simulation et de l'évaluation de leur pire temps d'exécution par OTAWA.

4.1.1 Choix des benchmarks

Pour nos expérimentations, nous avons recherché des benchmarks "temps réel" au sens où leur pire temps d'exécution est calculable. Pour cela, ils ne doivent pas contenir de boucles infinies ou dont la borne ne peut pas être déterminée ou estimée. De plus, CarCore étant dépourvu d'unité de calcul flottant, nous avons éliminé les programmes qui comportaient de tels calculs. En effet, les calculs flottants sur un processeur qui ne les supporte pas nativement se font grâce à des fonctions d'émulation qui sont insérées par le compilateur et dont on ne maîtrise pas le code source. Ces fonctions comportent des boucles qu'il est souvent difficile de borner avec précision simplement en désassemblant le code objet de la fonction.

Nos recherches ont permis de faire une préssélection d'un petit nombre de suites de benchmarks :

- les SNU-RT[39] sont très simples et implémentent de petites fonctions de calcul (produit matriciel, transformée de Fourier rapide, ...). Ils sont diffusés par une équipe de l'Université de Singapour,
- les EEMBC[40] [81] ont été créés par un consortium d'industriels et ne sont disponibles que sous licence. Ils se présentent sous forme de "séries" thématiques (automotive, telecom, consumer...). Ce sont des programmes assez simples, du genre des SNU-RT,
- les MiBenchs [50] sont développés par l'université du Michigan. Nous n'avons pas pu calculer les informations de flots de certains d'entre eux,
- l'université de Mälardalen maintient une suite de benchmarks [89] qui incluent les SNU-RT,
- PapaBench [88] est un code de contrôle d'un drône civil écrit par une équipe de l'ENAC et adapté dans notre équipe. Le code est assez simple, ce qui est intéressant est qu'il est constitué de plusieurs tâches. On calcule alors le pire temps d'exécution de chacune des tâches. Pas de jeux de données pour ce benchmark ce qui est un problème puisqu'on ne peut donc pas les simuler. Les tâches comportent beaucoup de calcul flottant.

Nous avons écarté les EEMBC parce qu'ils ne sont disponibles que sous licence et les MiBenchs parce que nous avons eu des problèmes de simulation et de détermination des informations de flots ainsi que PapaBench parce qu'ils compote trop de calcul flottant et qu'on ne dispose pas des jeux de données. Notre choix s'est donc naturellement porté sur les benchmarks de l'université de Mälardalen qui incluent les SNU-RT.

Les benchmarks de L'université de Mälardalen

Le groupe de recherche de l'université de Mälardalen sur le pire temps d'exécution des programmes maintient une longue liste de benchmarks largement utilisée (par exemple lors du WCET challenge 2006) pour évaluer et comparer les méthodes et outils d'analyse du pire temps d'exécution des programmes.

Ces benchmarks sont collectés à partir de plusieurs sources universitaires et industrielles du monde entier. Chaque programme est fourni, entre autre, avec son code source en langage C, et les informations de flots.

Le tableau 4.1 présente les programmes de la suite proposée par l'université de Mälaradalen et quelques unes de leurs caractéristiques. Nous avons écarté ceux qui présentaient des calculs flottants ainsi que ceux qui faisaient appel à des routines ex-

[39] Singapour National University-Real Time
[40] Embedded Microprocessor Benchmark Consortium

Nom	Description
bs	Recherche binaire dans un tableau de 15 éléments
bsort100	Programme de tri à bulle
fdct	Transformation rapide par cosinus discret
fibcall	suite de Fibonacci
jfdctint	Transformation par cosinus discret sur un bloc de 8x8 pixels
ns	Recherche dans un tableau multidimensionnel
nsichneu	Simulation d'un réseau de Pétri étendu

TAB. 4.1 – Quelques caractéristiques des benchmarks de l'université de Mälardalen

térieures dont nous ne contrôlons pas l'exécution. Nous avons en outre rencontré des problèmes avec un certain nombre de ces benchmarks lors de la simulation. Dans le cas le plus fréquent, les programmes utilisaient des instructions qui n'étaient pas prises en compte par le simulateur de CarCore fourni par l'équipe de l'université d'Augsbourg. Finalement, nous avons pu utiliser les programmes suivants : bs, bsort100, fdct, fibcall, insertsort, jfdctint, ns et nsichneu.

4.1.2 Préparation des benchmarks

Afin d'être utilisés pour notre évaluation, ces programmes ont été compilés avec le compilateur gcc pour produire des binaires pour TriCore. Ce compilateur s'exécute sur une architecture x86 et produit des binaires pour le TriCore (cross-compilateur). Il nous a été fourni par l'université d'Augsbourg avec le simulateur de CarCore.

Une fois les programmes compilés, il faut déterminer les informations de flots. OTAWA supporte deux formats de fichiers d'informations de flots : le format f4 (Flow Facts File Format[41]) et le format ffx (flow facts XML format[42]). Ces deux formats ont été proposés par notre équipe dans le cadre du déveoppement d'OTAWA [13].

Lorsque nous avons commencé nos mesures, seul le format f4 était pris en compte, nous avons donc utilisé ce format. L'extension usuelle des fichiers à ce format est "`.ff`". Ces fichiers sont des fichiers texte. Des commentaires peuvent y être intégrés. Chaque ligne du fichier se termine par un point-virgule. Le tableau 4.2 résume les commandes utilisables dans ce format.

[41] Format de fichier d'informations de flot
[42] format XML d'informations de flot

Mot-clé	Déscription
`loop`	Introduit une boucle. Le mot-clé est suivi de l'adresse de la boucle et du nombre d'itérations. Si le nombre d'itérations n'est pas fixe, il peut être remplacé par le nombre maximum d'itérations (précédé du mot clef max) et/ou le nombre total d'itérations pour le programme entier (précédé du mot clef total).
`loop ... in`	Permet de désigner une boucle comme imbriquée dans d'autres. Les options max et total sont aussi utilisables.
`cheksum`	Introduit la somme de contrôle du programme.
`return`	Suivi d'une adresse, marque l'instruction désignée comme étant un retour de sous-programme.
`noreturn`	Suivi d'une adresse, indique qu'une fonction est sans retour.
`ignorecontrol`	Suivi d'une adresse, force OTAWA à ignorer l'effet de l'instruction de contrôle désigné par l'adresse.
`multibranch ... to`	suivi d'une liste d'adresses, indique un branchement à cible multiples et la liste des adresses cibles.
`nocall`	suivi d'une adresse de fonction, indique à OTAWA de traiter la fonction désignée par son adresse comme si ce n'était pas une instruction de contrôle.
`preserve`	Suivi d'une adresse, assure que le chargeur d'informations de flot ne modifiera pas l'instruction désignée par l'adresse.

TAB. 4.2: Les attributs possibles dans le format f4

Les adresses peuvent être données en décimal, en octal (en les préfixant de `0`), en hexadécimal (en les préfixant de `0x` ou `0X`) ou bien en binaire (en les préfixant de `0b` ou `0B`). Elles peuvent être absolues ou sous la forme d'un décalage par rapport à une étiquette, par exemple : `"main" + 0x80`.

Le fichier commence par une somme de contrôle qui permet à OTAWA de reconnaître et de signaler tout programme qui aurait été recompilé et qui pourrait ne pas correspondre au fichier d'informations de flots. La directive `noreturn` est souvent utilisée pour la fonction `_exit` de la libC. La directive `multibranch` permet de traiter des pointeurs de fonctions ou des structures algorithmiques comme les switch . . .case,

```
checksum "bs.elf" 0x376b99b1;

// Function _start
loop "_start" + 0x10c ?;
loop "_start" + 0x13a ?;
  loop "_start" + 0x152 ?;
loop "_start" + 0x188 ?;
  loop "_start" + 0x1a2 ?;

// Function _exit
loop "_exit" + 0x2 ?;

// Function binary_search
loop "binary_search" + 0x12 ?;
```

FIG. 4.1 – Fichier d'informations de flots produit par mkff pour le programme bs

qui sont souvent traduits par les compilateurs par des instructions de contrôle à cibles multiples indirects.

La directive `nocall` est utilisée pour éviter l'appel à des fonctions non désirées.

`mkff` est une commande d'OTAWA qui produit un fichier canevas qu'il suffit de remplir ensuite. La figure 4.1 reproduit le fichier produit pour le programme `bs`. Pour chaque fonction, toutes les boucles sont identifiées par un décalage par rapport à l'étiquette qui introduit la fonction. Il faut remplacer le `?` par le nombre d'itérations de la boucle. L'indentation dans le fichier correspond à l'imbrication des boucles dans le programme.

Pour déterminer le nombre d'itérations des boucles, on utilise les informations données avec le programme, son code source et son graphe de flot de contrôle. Ce dernier est obtenu facilement grâce à `dumpcfg`, un autre programme fourni par OTAWA.

La détermination des bornes de boucles se fait à la main en analysant le code assembleur des programmes. Les valeurs fournies avec les benchmarks et celles contenues dans le code source sont souvent justes mais le compilateur fait parfois des optimisations qui modifie le nombre d'itérations. Parfois même, des modifications plus profondes de la structure du programme sont possibles. Il est donc nécessaire de vérifier toutes les valeurs.

Nom	Nombre de blocs de base	Taille du plus grand bloc de base	Proportion d'instructions mémoire	Proportion d'instructions de contrôle
bs	14	15	36,49%	16,22%
bsort100	25	5	38,40%	16,80%
fdct	15	328	55,02%	2,07%
fibcall	10	12	52,38%	21,43%
jfdctint	17	249	50,45%	2,52%
ns	24	37	34,29%	18,10%
nsichneu	757	52	35,81%	6,16%

TAB. 4.3 – Caractéristiques statiques des programmes de test

4.1.3 Analyse des benchmarks utilisés

Nous nous sommes intéressés aux caractéristiques des programmes de test que nous avons utilisé. Les éléments qui nous ont paru pertinents sont le nombre de blocs de base, leur taille, la proportion d'instructions mémoire et d'instructions de contrôle dans les programmes. Le tableau 4.3 donne les caractéristiques statiques des benchmarks.

On remarque leur relative homogénéité. Seuls quelques programmes comprennent des valeurs éloignées du groupe comme nsichneu avec ses 757 blocs de base, fdct et jfdctint avec des blocs de base qui comportent respectivement jusqu'à 328 et 249 instructions.

Nom	proportion des instructions "normales"	Proportion d'instructions mémoire	Proportion d'instructions de contrôle
bs	34,62%	26,92%	38,46%
bsort100	34,27%	40,90%	24,83%
fdct	38,24%	43,43%	18,34%
fibcall	38,94%	33,63%	27,43%
jfdctint	25,63%	23,74%	50,63%
ns	18,24%	35,71%	46,05%
nsichneu	12.36%	35,96%	51,69%

TAB. 4.4 – Caractéristiques dynamiques des programmes de test

Dans le tableau 4.4, ce sont les statistiques dynamiques qui sont produites, c'est à dire les proportions d'instructions réellement exécutées sur le chemin de contrôle qui produit le pire temps d'exécution.

Ces statistiques sont mises en rapport avec les résultats expérimentaux dans la sec-

tion 4.2. Elles aident à comprendre d'ou viennent les imprécisions du modèle.

4.2 Résultats

L'exploitation du parallélisme de tâche rendu possible par les processeurs modernes peut fournir aux systèmes temps réels les performances dont ils ont besoin. Il est cependant nécessaire de pouvoir calculer le pire temps d'exécution de chaque tâche. Ce temps dépend de la micro-architecture du processeur qui exécute les tâches. La modélisation du parallélisme de tâche apporte des difficultés supplémentaires discutées dans la section 2.4.

Nous avons proposé d'utiliser les graphes d'exécution pour modéliser l'exécution des blocs de base des tâches par le processeur CarCore, qui est conçu pour isoler temporellement une tâche temps réel de l'influence des autres tâches co-exécutées par le processeur. Nous pouvons ensuite utiliser la méthode d'évaluation du pire temps d'exécution des tâches par énumération des chemins implicites.

Dans cette section, nous proposons une évaluation de notre modèle en termes de précision. Nous voulons estimer le pessimisme des WCETs calculés.

4.2.1 Évaluation du modèle de CarCore

L'étape suivante est la simulation des programmes de tests choisis. Nous avons utilisé le simulateur de CarCore qui nous a été fourni par l'équipe du professeur Ungerer de l'université d'Augsbourg. C'est une version de développement mais elle est suffisamment complète pour nos mesures. Elle fournit en particulier des statistiques suffisantes sur l'exécution des programmes, tant globales qu'au niveau de chaque étage à chaque cycle d'exécution. Ces simulations nous ont permis de mesurer les temps d'exécututions réels des blocs de base.

D'autre part, nous avons évalué, grâce à OTAWA, les coûts de ces même blocs de base. Puis nous avons comparé les temps évalués et les temps mesurés. D'abord globalement, c'est-à-dire, en prenant en compte les informations de flot, nous avons estimé le pire temps d'exécution de chaque programme de test, et nous l'avons comparé avec le temps d'exécution mesuré. Puis, nous avons travaillé à un grain plus fin, celui des blocs de base.

benchmark	bs	bsort100	fdct	fibcall	jfdctint	ns	nsichneu
surestimation	5,1%	4,.5%	0,5%	5,5%	5,7%	6,9%	1,1%

TAB. 4.5 – surestimation du pire temps d'exécution estimé par rapport au temps mesuré

Les résultats globaux

Le tableau 4.5 présente les résultats globaux obtenus sur les programmes de test que nous avons utilisé. La seconde ligne présente la surestimation, c'est-à-dire le rapport entre la différence (en nombre de cycles) entre le pire temps d'exécution estimé et le pire temps d'exécution mesuré par simulation d'une part et le temps mesuré par simulation d'autre part : $surestimation = \frac{(temps_{estim} - temps_{mesur})}{temps_{mesur}}$.

Ces résultats confirment qu'aucun pire temps d'exécution n'a été sous-estimé, mais on note une surestimation pour tous les programmes de tests de 3,93% en moyenne. Cette surestimation, bien que faible, peut laisser penser que notre modèle n'est pas assez fidèle au matériel qu'il représente. Nous avons donc cherché les causes de ces surestimations.

Pour cela, nous avons mis en parallèle ces résultats globaux avec les caractéristiques des programmes de tests résumés dans le tableau 4.3. Il semble que nos résultats sont meilleurs sur les programmes qui comportent de plus grands blocs de base ainsi que sur ceux qui ont une plus faible proportion d'instructions de contrôle.

L'examen des statistiques dynamiques, résumées dans le tableau 4.4, n'apporte pas beaucoup plus d'informations. Le programme de test le moins surévalué (`fdct`) est celui qui comporte le moins d'instructions de contrôle, mais nsichneu qui est le second dans l'ordre croissant de surestimation est un de ceux qui en comportent le plus. On ne peut donc pas dégager de tendance de ces statistiques.

Il est donc probable que les effets de l'imprécision de notre modèle soient masqués par des effets qui ne sont pas liés à la modélisation. Par exemple le nombre d'exécutions d'un bloc de base a un impact fort sur la précision du pire temps d'exécution évalué. Pour nous libérer de ces effets externes, nous nous sommes intéressés directement à la précision de l'évaluation des blocs de base.

Les résultats au niveau des blocs de base

L'analyse a porté sur 753 blocs de base appartenant aux 7 benchmarks. Sur l'ensemble des blocs considérés, 547 ne sont pas surévalués ce qui représente 72,64 % de l'ensemble des blocs. La surestimation moyenne des blocs de base est de 2,68 %. Sur

Benchmark	surestimation d'1 cycle	surestimation de 3 cycles	surestimation de 4 cycles
bs	12	1	2
bsort100	7	1	0
fdct	7	1	0
fibcall	1	5	0
jfdctint	9	0	1
ns	2	1	4
nsichneu	2	1	0

TAB. 4.6 – Statistiques de surestimation des blocs de base des programmes de test

les 206 blocs surestimés, 172 le sont d'un cycle, 26 le sont de 3 cycles et 8 le sont de 4 cycles. Nous avons donc analysé les blocs surestimés pour connaître les causes de ces surestimations. Nous en avons trouvé deux. L'une correspondant aux blocs surestimés d'un cycle, l'autre à ceux surestimés de trois cycles. Les blocs surestimés de quatre cycles subissent les deux effets cumulés. Le tableau 4.6 donne le détail pour chaque programme de test. L'unité est le nombre de blocs de base.

Les blocs surestimés d'un cycle

Lors d'un chargement mémoire, l'unité de chargement fait une requête à la mémoire sur l'adresse demandée par l'instruction. La donnée est renvoyée et mise dans un tampon de l'unité de gestion mémoire. Or le bus de donnée a une largeur de 64 bits. A chaque requête, 64 bits sont renvoyés de la mémoire vers le processeur.

Chaque tâche dispose d'un tampon de 64 bits dans l'unité de gestion de la mémoire. Lors de la requête mémoire suivante, si la donnée demandée se trouve dans le tampon, elle est directement utilisée par le processeur sans qu'une requête mémoire ne soit nécessaire. C'est ainsi qu'on gagne un cycle sur certains chargements mémoires.

La modélisation de ce mécanisme est rendue difficile par le fait que les accès mémoires sont le plus souvent indirects. Les adresses des données accédées ne sont pas directement codées dans les instructions mais elle sont calculées et stockées dans des registres avant l'accès. Il n'est donc pas possible de savoir si deux adresses contiguës vont être lues successivement en ne regardant que le code des instructions.

Pour nos programmes de test, ce cas est majoritairement rencontré lors d'accès à des tableaux. Les programmes de tests que nous avons utilisés et qui utilisent des tableaux sont : `bs`, `bsort100`, `fdct`, `jfdctint` et `ns`. Ils correspondent aux programmes dans lesquels cette surestimation revient le plus souvent. En dehors des tableaux, les accès à la pile peuvent occasionner cette situation.

Les blocs surestimés de 3 cycles

Dans un processeur dont la prédiction de branchement est statique, le processeur continue le plus souvent à charger les instructions en séquence lorsqu'un branchement est rencontré. Lorsque le branchement est pris, les tampons qui sont succeptibles de contenir des instructions du mauvais chemin sont vidés du pipeline et l'exécution reprend à l'adresse cible du branchement.

Dans CarCore, lorsque le processeur rencontre ce cas-là, si la fenêtre d'instructions contient déjà la cible du branchement, seules les instructions qui sont sur le mauvais chemin sont purgées. Il n'est donc pas nécessaire de recharger celles qui sont à la cible du branchement. C'est ainsi que le simulateur gagne trois cycles sur le pire temps d'exécution évalué. Un cycle est gagné car on sait si le branchement est pris à l'étage d'exécution, les deux autres correspondent à la traversée de l'étage de chargement et à la latence de la mémoire d'instruction pour la première instruction chargée après le branchement.

Cette situation se produit lorsque la cible d'un branchement en avant (sortie de boucle) est proche de celui-ci. Par exemple lorsque le corps d'une boucle est formé d'un petit nombre d'instructions.

Pour supprimer cette source d'imprécision, il faut savoir si les instructions à la cible du branchement se trouvent dans la fenêtre d'instructions ou pas. C'est à dire qu'il faut connaître la place libre dans la fenêtre d'instructions quelques cycles avant le branchement. Ceci dépend de deux paramètres : la taille de la fenêtre d'instructions et le chemin de contrôle pris précédemment dans le programme. En effet, le nombre d'emplacements libres dans la fenêtre d'instructions dépend des branchements rencontrés précédemment et qui ont entrainé la vidange de celle-ci. Ces vidanges libèrent de la place dans la fenêtre. Ce dernier paramètre est difficile à maitriser dans le cadre d'une analyse statique.

Nous avons donc pu déterminer que les sources de l'imprécision de notre modèle de CarCore étaient liées à des optimisations de l'architecture au chargement ainsi que lors des accès mémoires. Ces optimisations induisent un comportement dynamique que nous n'avons pas pu modéliser simplement dans le cadre d'une analyse statique. L'impact de ces imperfections sur la précision de l'estimation du pire temps d'exécution global dépend du nombre d'exécution des blocs de bases qu'elles impactent et donc du flot de contrôle lui-même. Pour les programmes de test que nous avons utilisé, cet impact reste faible.

Configuration	F-8	F-16	F-32	F-64	F-128
Taille	8 octets	16 octets	32 octets	64 octets	128 octets

TAB. 4.7 – Les configurations de tests pour la taille de la fenêtre d'instructions

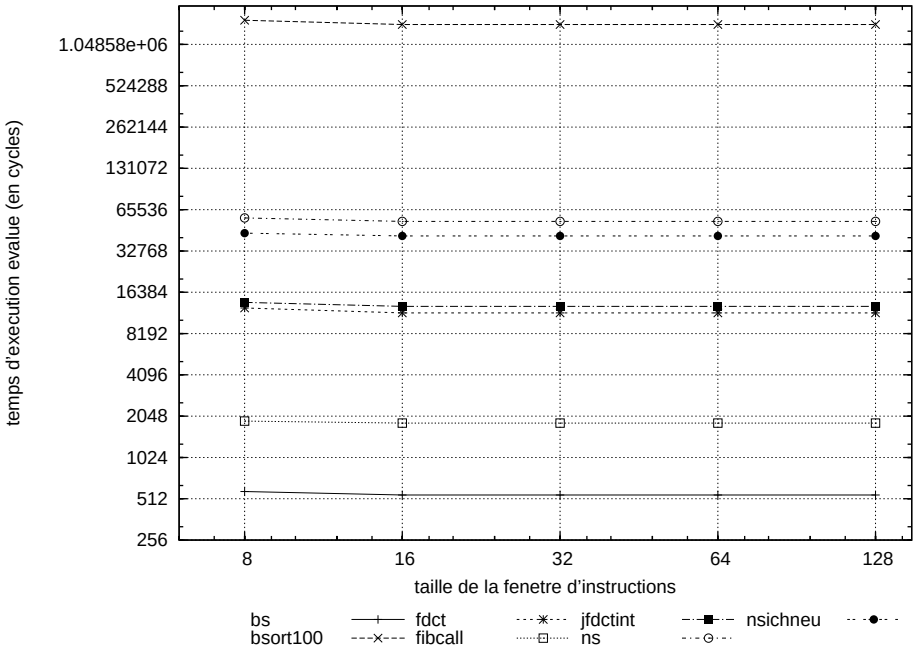


FIG. 4.2 – pire temps d'exécution évalué en fonction de la taille de la fenêtre d'instructions

4.2.2 Impact de quelques paramètres du processeur sur le pire temps d'exécution

Dans cette section, nous examinons l'impact de certains paramètres du pipeline sur le pire temps d'exécution, non pas en termes de précision de l'estimation, mais en termes d'allongement du temps d'exécution pire cas.

Nous sommes partis du processeur CarCore et avons considéré deux paramètres : la taille de la fenêtre d'instructions, et l'allongement du pipeline en divers endroits.

La taille de la fenêtre d'instructions

Nous avons considéré 5 configurations, donc 5 tailles de fenêtre d'instructions différentes indiquées dans la table 4.7. Ces résultats sont les pires temps d'exécution (en cycles) de chaque bloc de base de chaque programme de test.

La figure 4.2 montre, pour chaque programe de test, le pire temps d'exécution évalué en fonction de la taille de la fenêtre d'instructions. L'échelle est logarithmique sur

Augmentation (en cycles)	0	1	2	3	6	9	20	29
Nombre de blocs de base	717	7	2	4	128	1	2	1
Proportion	83.17%	0.81%	0.23%	0.46%	14.84%	0.11%	0.23%	0.11%

TAB. 4.8 – Répartition de l'augmentation du pire temps d'exécution des blocs de base dans la configuration F-8

les deux axes. On remarque que pour tous les benchmarks, les temps des configurations F-16 à F-128 sont identiques. les résultats sont les mêmes quand on regarde en détail au niveau des blocs de base. La taille de la fenêtre d'instruction n'a presque aucune influence sur le pire temps d'exécution. Cela signifie que l'étage d'ordonnancement, qui prend les instructions dans la fenêtre d'instructions pour les lancer dans le reste du pipeline, trouve la fenêtre d'instructions toujours approvisionnée à partir d'une fenêtre de 16 octets, ce qui correspond à des tailles allant de 4 à 8 instructions. Ceci s'explique par le fait que les instructions sont chargées par groupes de 64 bits et que la latence de ce chargement est d'un cycle. Or à chaque cycle l'étage d'ordonnancement ne peut lancer qu'au plus, deux instructions. Donc, dans le pire des cas, il sort à chaque cycle autant d'instructions qu'il en rentre. La fenêtre est donc toujours approvisionnée en instructions.

Le tableau 4.8 donne la répartition de l'augmentation du pire temps d'exécution estimé pour la configuration F-8. La première ligne donne le nombre de cycles supplémentaires par rapport aux autres configurations ; la seconde donne le nombre de blocs de base présentant cette augmentation et la troisième ligne montre la proportion de blocs de base concernés par rapport au nombre total de blocs testés. La grande majorité des blocs (83,17 %) ne présentent aucune augmentation.

Dans cette configuration, la fenêtre ne fait que 64 bits, ce qui est la taille du bus vers la mémoire d'instructions. A chaque cycle, la totalité des instructions chargées peuvent entrer dans la fenêtre à condition que celle-ci ait été totalement vidée par l'ordonnancement des instructions qui l'occupaient. Ceci ne correspond qu'au cas où deux instructions de 32 bits sont lancées dans leur pipeline respectifs (la première dans le pipeline entier et la seconde dans le pipeline adresse). A chaque fois qu'on a un ordonnancement différent de celui-ci, la totalité des instructions chargées ne peut pas entrer dans la fenêtre d'instructions. Ceci n'a d'impact que lorsque, au cycle suivant, une instruction qui aurait du être lancée n'est pas dans la fenêtre. Le processeur doit alors attendre le cycle suivant que l'instruction manquante soit chargée.

Ceci pourrait induire une grosse augmentation du pire temps d'exécution estimé

dans la configuration F-8 par rapport aux autres configurations. Cependant, plusieurs facteurs atténuent ce phénomène. Les opérations à latence longue, comme les branchements ou les opérations mémoires, sont nombreuses et permettent au chargement de se faire pendant leur latence. D'autre part, les deux pipelines ne sont pas toujours utilisés par la même tâche : la probabilité de présence d'une seule instruction dans la fenêtre est plus grande que la probabilité de présence de deux instructions.

Dans CarCore, la taille choisie est de 48 octets, le problème mis en évidence pour la configuration F-8 ne se pose donc pas. Pour la tâche temps réel, 16 octets auraient sufit, mais les autres tâches peuvent profiter d'une taille plus grande.

La taille du pipeline

Pour évaluer l'impact de la taille du pipeline sur le pire temps d'exécution estimé, nous avons considéré l'ajout d'étages en plusieurs points du pipeline de CarCore. Nous avons construit 4 configurations, correspondant à 4 points du pipeline ou nous avons fait des ajouts d'étages :

- la configuration *P-1* ajoute les étages entre l'étage de chargement et la fenêtre d'instructions,
- la configuration *P-2* ajoute les étages entre la fenêtre d'instructions et l'étage d'ordonnancement,
- la configuration *P-3* ajoute les étages entre l'étage d'ordonnancement et l'étage d'exécution dans les 2 pipelines,
- la configuration *P-4* ajoute les étages entre l'étage d'exécution et celui d'écriture des registres dans les 2 pipelines.

Pour chacune de ces configurations, nous avons considéré l'ajout d'1, 2 et 3 étages. Les étages que nous ajoutons ne font aucun traitement particulier, ils se contentent de laisser passer les instructions.

Les figures 4.3 à 4.6 présentent les courbes d'augmentation des pires temps estimés des programmes de tests. L'augmentation (A) est calculée comme le quotient de la différence entre le temps d'exécution estimé pour une configuration et un nombre d'étages ajoutés (t_{config}) et le pire temps d'exécution estimé pour le processeur CarCore sans modification ($t_{CarCore}$) sur le pire temps d'exécution estimé pour le processeur CarCore sans modification : $A = 100 * (t_{config} - t_{CarCore})/t_{CarCore}$. Ils sont donnés en pourcentages. Par exemple, dans la configuration P-1, pour le programme bs, la valeur des ordonnées pour l'abscisse 1 (1 étage ajouté) est calculée en divisant le pire temps d'exécution estimé pour l'ajout d'un étage entre l'étage de chargement et la fenêtre

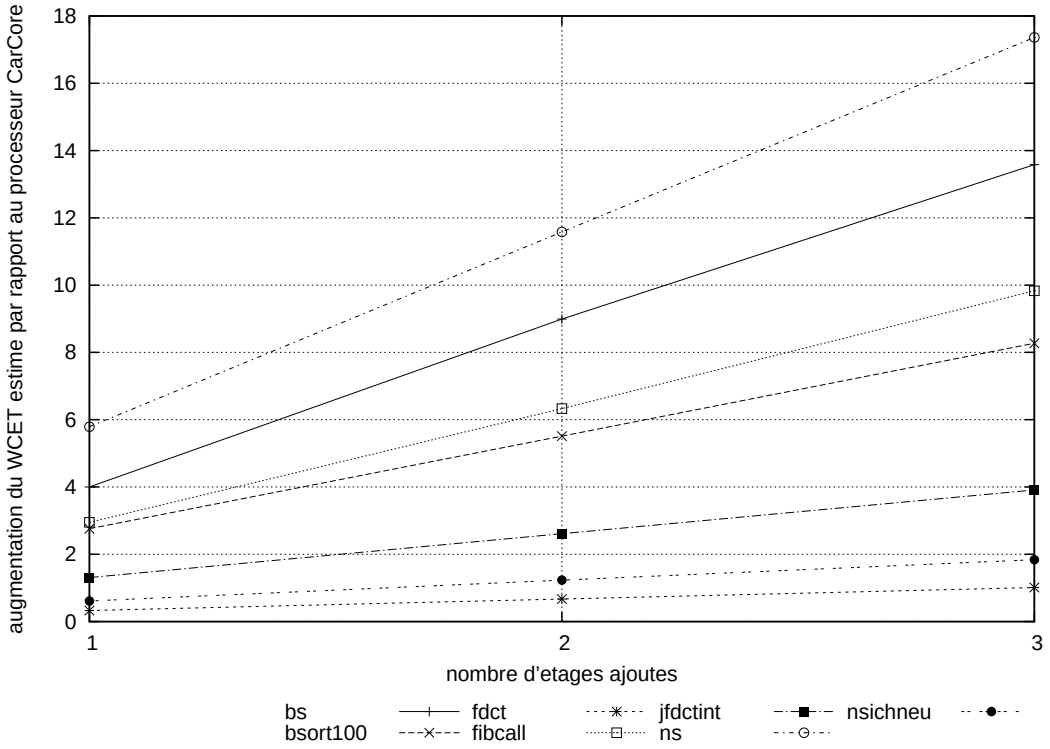


FIG. 4.3 – Augmentation du pire temps d'exécution estimé pour la configuration P-1

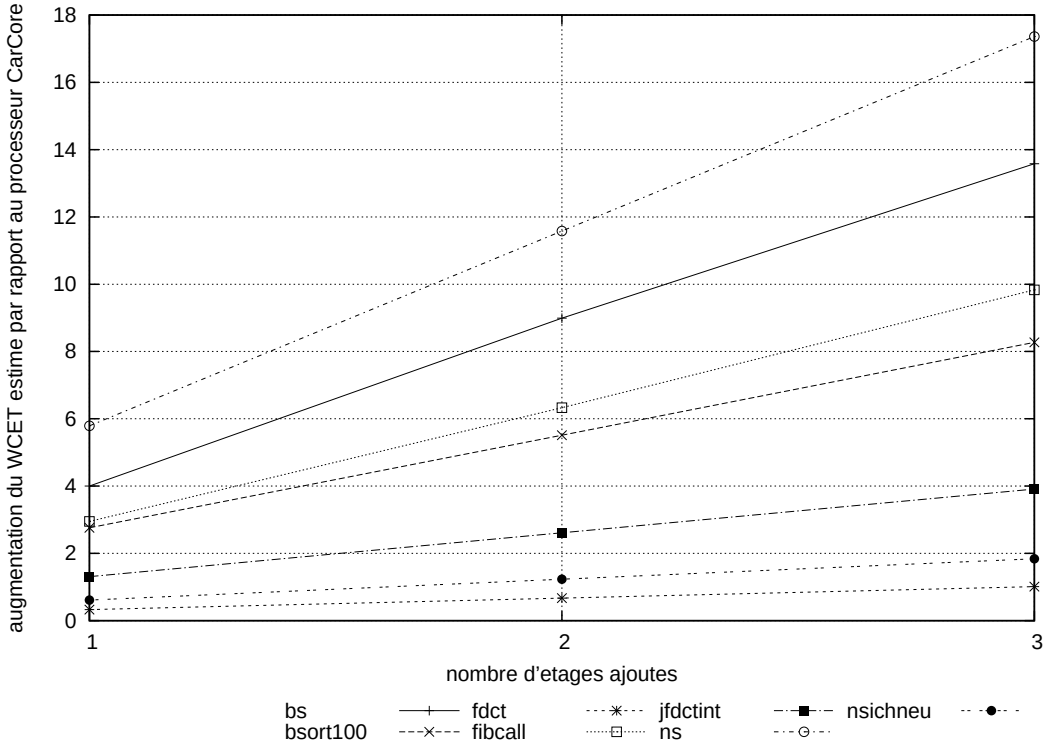


FIG. 4.4 – Augmentation du pire temps d'exécution estimé pour la configuration P-2

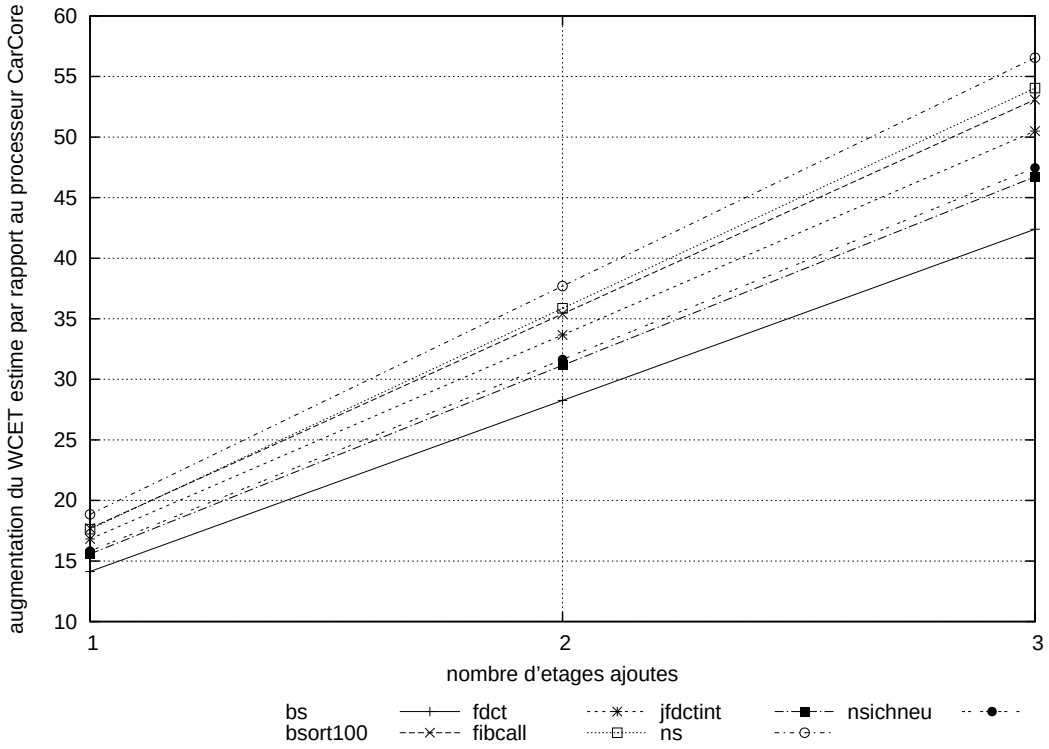


FIG. 4.5 – Augmentation du pire temps d'exécution estimé pour la configuration P-3

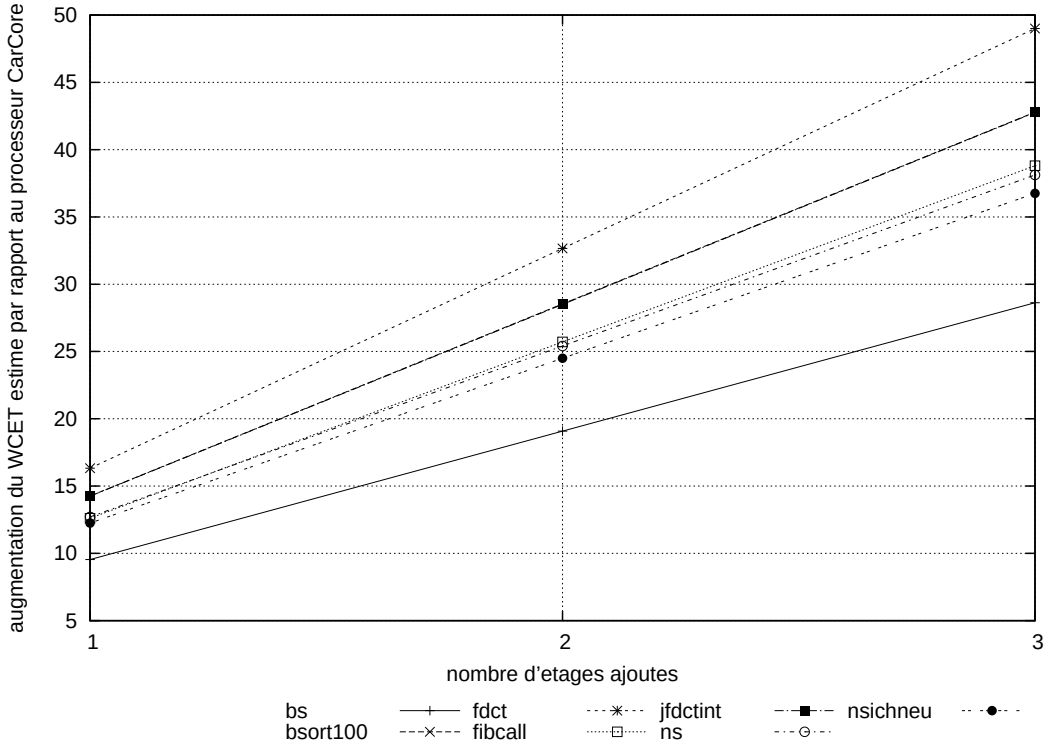
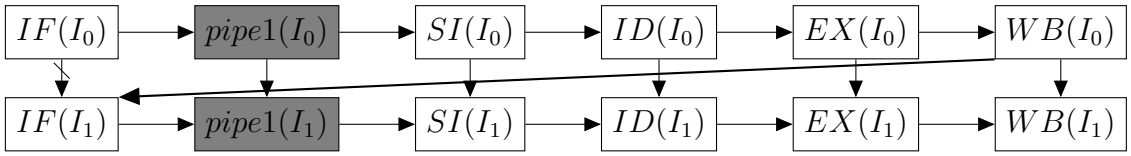
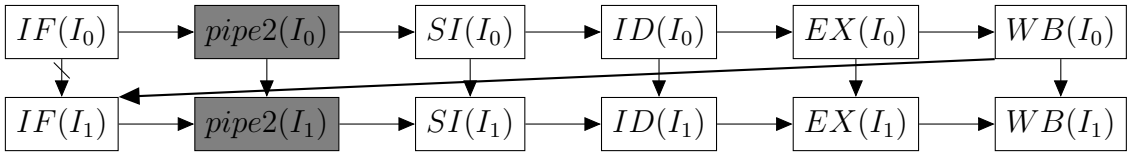


FIG. 4.6 – Augmentation du pire temps d'exécution estimé pour la configuration P-4



(a) extrait de graphe d'exécution pour l'ajout d'un étage dans la configuration P-1



(b) extrait de graphe d'exécution pour l'ajout d'un étage dans la configuration P-2

FIG. 4.7 – Pour les configurations P-1 et P-2, les graphes d'exécution sont les mêmes

d'instructions par le pire temps d'exécution estimé pour le processeur CarCore et multiplié par cent. Le pire temps estimé est le pire temps global estimé pour la tâche. Il est calculé en faisant la somme des pire coûts des blocs de base qui sont sur le chemin critique, pondérés par leur nombre d'exécution.

De ces courbes, on remarque que le pire temps d'exécution croit linéairement avec le nombre d'étages ajoutés dans tous les cas. L'ajout de chaque étage ajoute toujours le même nombre de cycles au pire temps d'exécution estimé car, d'une part chaque étage a une latence constante et identique (un cycle), d'autre part l'attente des instructions bloquées dans la fenêtre d'instructions en attente d'être lancées est linéairement proportionnelle au nombre d'étages que les instructions qui les bloquent doivent traverser pour résoudre le blocage.

On remarque aussi que les figures 4.3 et 4.4 sont identiques. Ceci s'explique en examinant les graphes d'exécution : Les arcs horizontaux, qui modélisent la progression des instructions dans le pipeline, et verticaux, qui représentent l'ordre d'exécution des instructions, sont liés à tous les étages, y compris les étages ajoutés. Les autres arcs qui relient deux lignes représentent les dépendances et ne touchent jamais les étages ajoutés. Ils sont liés soit au premier étage de chargement lorsqu'ils représentent un branchement pris ou la taille de la fenêtre d'instructions, soit à l'étage d'ordonnancement lorsqu'ils représentent des branchements non pris ou des dépendances de données. Ainsi, que les étages soient ajoutés entre le chargement et la fenêtre d'instructions ou entre celle-ci et l'étage d'ordonnancement, ils ajoutent toujours le même délai au temps d'exécution.

La figure 4.7 illustre cette situation. La partie 4.7(a) montre un extrait de graphe d'exécution pour la configuration P-1 et la partie 4.7(b) présente le même extrait (même

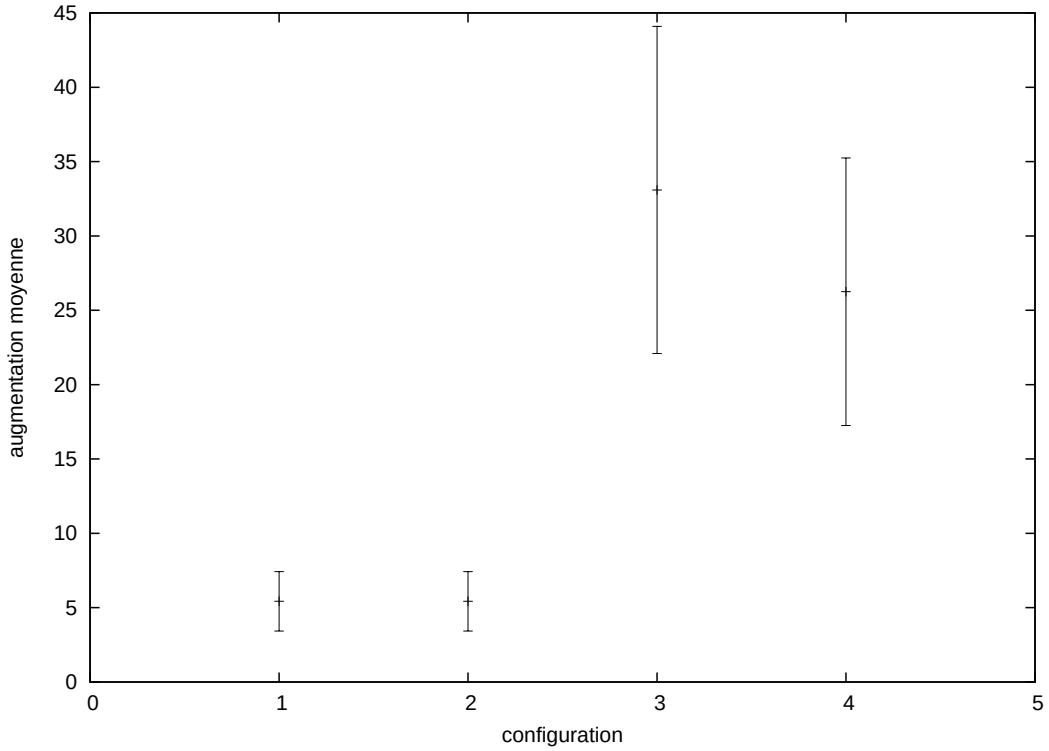


FIG. 4.8 – Augmentation moyenne pour chaque configuration

séquence d'instructions) pour la configuration P-2. Les deux graphes sont identiques. Les pires temps d'exécution évalués sont donc les mêmes.

La figure 4.8 montre, pour chaque configuration P-1 à P-4, l'augmentation moyenne pour l'ensemble des programmes de test et quel que soit le nombre d'étages ajoutés. La moyenne est la valeur centrale, la barre associée à chaque moyenne donne l'écart moyen à la moyenne. La position de l'ajout des étages a donc une influence sur le pire temps d'exécution estimé. C'est la configuration P-3 qui augmente le plus le pire temps estimé.

Pour comprendre cette influence et comment elle entraîne de tels écarts, nous avons examiné pour chaque programme de test, le comportement de l'augmentation moyenne pour chaque configuration. La figure 4.9 résume ce comportement. Les abscisses sont les configurations (P-1 à P-4) et les ordonnées sont les augmentations moyennes par rapport au pire temps estimé pour le processeur CarCore sans modification.

Nous avons confronté ces courbes aux statistiques des programmes de test (tableau 4.3). Pour la configuration P-3, on remarque que les configurations qui ont la plus grande augmentation (ns, fibcall et bsort100) sont celles qui comportent la plus grande proportion d'instructions de contrôle. Alors que pour la configuration P-3, ce

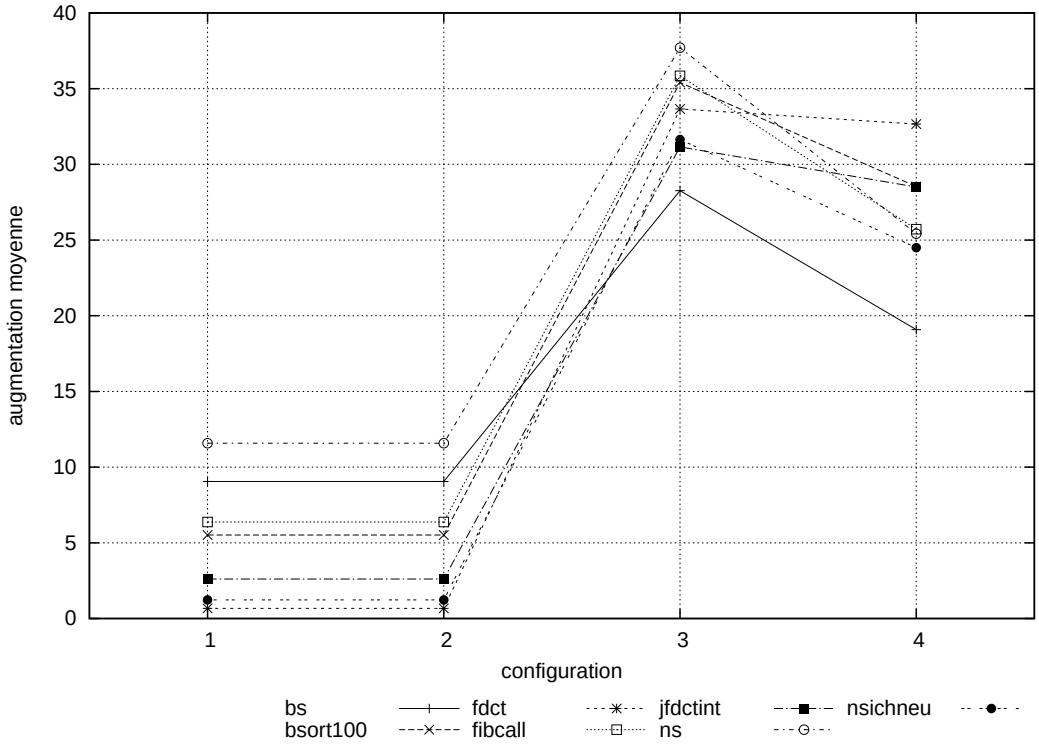
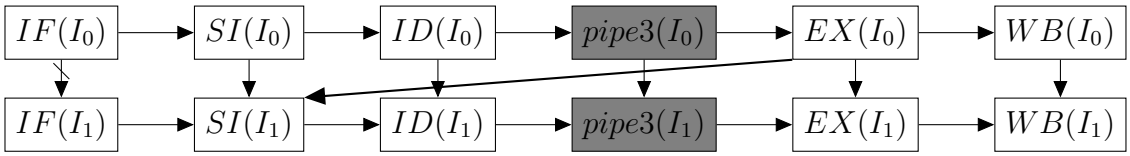


FIG. 4.9 – Augmentation moyenne pour chaque programme de test

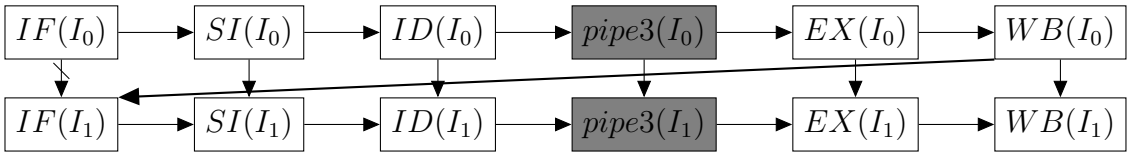
sont celles qui ont les plus grandes proportions d'instructions mémoire (fdct, jfdctint et fibcall).

Pour ces deux configurations, le arcs des graphes d'exécution qui permettent de comprendre ces résultats sont ceux qui expriment les dépendances de contrôle et les dépendances structurelles lors d'un chargement mémoire. Ils sont montrés dans les figures 4.10 et 4.11. Dans ces deux configurations, l'ajout d'étages n'a pas seulement un effet sur la propagation des instructions dans le pipeline (arcs horizontaux) et l'ordre d'exécution (arcs verticaux), mais aussi sur les latences induites par ces dépendances. Ceci explique que les augmentations pour les configurations P-3 et P-4 soient plus importantes que pour P-1 et P-2.

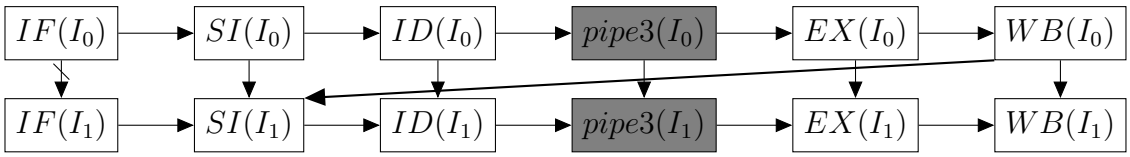
D'autre part, dans la configuration P-4, les étages ajoutés le sont après le nœud d'exécution, la latence induite pour les opérations mémoires n'est donc pas affectée (figure 4.11(a)). C'est la raison pour laquelle l'augmentation moyenne est moindre dans la configuration P-4 que dans la configuration P-3.



(a) chargement mémoire dans la configuration P-3

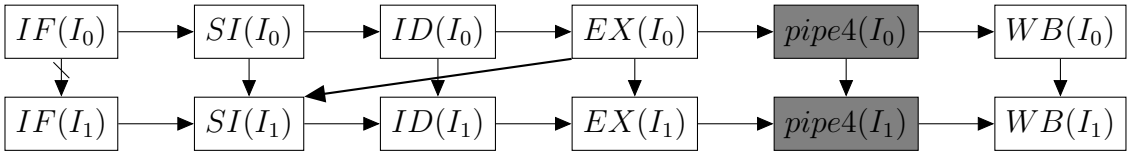


(b) Branchement pris dans la configuration P-3

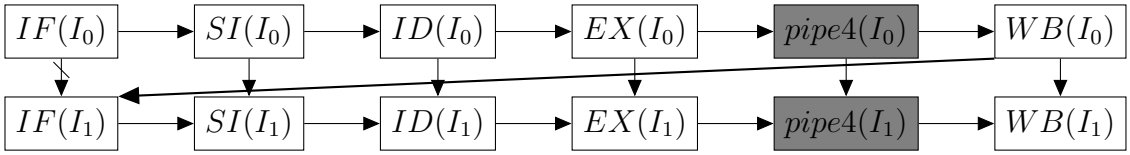


(c) branchement non pris dans la configuration P-3

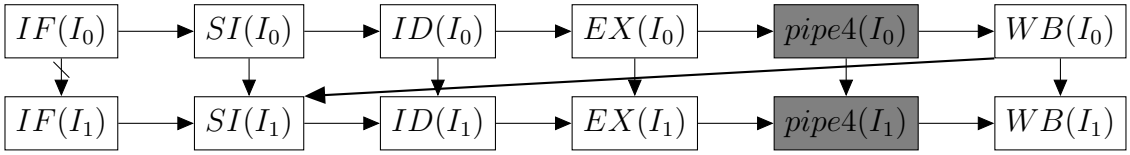
FIG. 4.10 – Impact de l'ajout d'étages, configuration P-3



(a) chargement mémoire dans la configuration P-4



(b) Branchement pris dans la configuration P-4



(c) branchement non pris dans la configuration P-4

FIG. 4.11 – Impact de l'ajout d'étages, configuration P-4

4.3 Conclusion

Le choix des programmes de test a posé problème : Il n'a pas été facile de trouver, parmi ceux communément utilisés dans le monde de l'informatique embarquée et du temps réel, des programmes de test dépourvus de calculs flottants, dont on pouvait borner les boucles à la main et dont toutes les instructions étaient reconnues par le simulateur de CarCore. Le calcul des bornes des boucles, lors de la préparation de l'évaluation du pire temps d'exécution, est un travail long, fastidieux et source d'erreurs. Des outils comme oRange permettent maintenant de faciliter cette tâche.

L'évaluation de notre modèle nous a permis de déterminer qu'il n'induisait jamais une sous-estimation du pire temps d'exécution, ce qui est une condition nécessaire pour son utilisation dans le cadre du temps-réel strict. Nous avons aussi pu constater que les surestimations obtenues au niveau des blocs de base sont limitées, ce qui est important pour le dimentionnement des systèmes. Les causes de ces surestimations sont liées aux aspects dynamiques de l'exécution des instructions de branchement et des opérations d'accès à la mémoire. Les phénomènes dynamiques qui se produisent sont difficiles à modéliser plus finement.

Nous avons aussi étudié l'impact de la taille de la fenêtre d'instructions ainsi que la taille du pipeline sur le pire temps d'exécution. Nous avons mis en évidence que l'emplacement des étages ajoutés au pipeline a un effet important sur l'augmentation du pire temps d'exécution. Ceci peut aider lors de la mise au point des microarchitectures, en particulier pour les extensions que nous suggérons dans la section 5.2.

La quantité de données produites lors des évaluations (de l'ordre de la centaine de giga octets) a rendu difficile leur analyse ainsi que leur expression sous forme de résultats synthétiques, même en ayant recours à des scripts. En particulier, la détermination des causes des surestimations, et l'impact de l'ajout d'étages dans les configurations *P-3* et *P-4* ont nécessité une analyse fine des traces de simulations et des graphes d'exécution produits.

Chapitre 5

Conclusion et perspectives

5.1 Résumé du travail accompli

Les systèmes temps réel ont besoin de puissance et de prévisibilité. Le parallélisme de tâches peut fournir de la puissance, parfois au prix de la prévisibilité. Son exploitation se fait dans deux classes de processeurs : les processeurs multicœurs et les processeurs multiflots. Ces derniers permettent à plusieurs flots de contrôle d'utiliser un même cœur en partageant ses structures internes.

La prévisibilité nécessaire aux systèmes temps réel se traduit par le besoin de connaître, a priori, le pire temps d'exécution des tâches. Ce temps dépend du jeu de données en entrée de la tâche et de l'architecture matérielle qui l'exécute. Il existe deux classes de méthodes pour évaluer le pire temps d'exécution d'une tâche : les méthodes dynamiques et les méthodes statiques. Ces dernières sont plus adaptées au temps réel strict, pourlequel on ne peut se permettre aucune violation des contraintes temporelles, parce qu'elles permettent de garantir que le pire temps d'exécution ne sera pas sous-évalué. Les méthodes statiques consistent à analyser les flots de contrôle et à trouver celui dont l'exécution sera la plus longue. Cela nécessite une modélisation de l'architecture qui exécute les tâches.

CarCore est un processeur multiflots simultanés qui a été conçu par l'équipe du professeur Ungerer de l'université d'Augsbourg (Allemagne). Il peut exécuter jusqu'à 4 tâches simultanément, en assurant l'indépendance temporelle de l'une d'elles, c'est à dire en assurant que les autres n'induisent pas de délai sur son temps d'exécution. Il est muni de 2 pipelines de 5 étages, dont les 2 premiers sont communs. L'un des pipelines est utilisé pour les opérations mémoires et l'autre pour les opérations entières. Les instructions sont exécutées dans l'ordre du programme. Chaque étage a une latence d'un

cycle. Les accès mémoire sont exécutés en deux phases et les instructions complexes sont microcodées pour éviter les blocages du pipeline.

Les graphes d'exécution permettent de modéliser l'exécution des blocs de base des tâches dans le processeur. Ils sont utilisés dans le cadre de l'analyse statique lors de l'évaluation du pire temps d'exécution des programmes. Il s'agit de graphes orienté dont les nœuds sont des couples (étage, instruction) et dont les arcs modélisent les liens de précédence entre les nœuds. Une précédence stricte est représentée par des arcs pleins alors que les arcs barrés montrent que les nœuds ainsi reliés peuvent s'exécuter simultanément ou dans l'ordre. Un parcours de ces graphes permet de déterminer le pire temps d'exécution des blocs de base dont ils modélisent l'exécution.

Nous avons utilisé les graphes d'exécution pour évaluer le pire temps d'exécution de tâches exécutées par CarCore. Nous avons modélisé la taille variable des instructions, le double pipeline et l'ordonnancement des instructions ainsi que les accès mémoires qui se font en deux phases et les microinstructions.

Nous avons ensuite implanté notre modèle en utilisant OTAWA, une plateforme qui fournit les outils nécessaires à l'évaluation du pire temps d'exécution de tâches. Pour cela, nous avons d'abord utilisé GLISS pour créer un module qui permet à OTAWA de comprendre le jeu d'instructions du TriCore d'Infineon utilisé par CarCore. Puis Nous avons écrit un chargeur spécifique pour OTAWA qui utilise la bibliothèque générée par GLISS et enfin nous avons modifié le module qui construit les graphes d'exécution pour que les graphes construits soient conformes à notre modélisation.

Pour évaluer la précision de notre modèle, nous avons choisi d'utiliser un sous-ensemble des programmes de tests de la suite de l'université de Malardälen. Nous avons comparé le pire temps d'exécution que nous avons évalué avec OTAWA avec celui obtenu par simulation. Le simulateur de CarCore nous a été fourni par l'équipe de l'université d'Augsbourg. Nos résultats ont montré que notre surestimation est de 3,93% en moyenne. Un examen plus fin au niveau des blocs de base nous a permis d'identifier deux causes de surévaluation liées au chargement des instructions lors des branchements et aux lectures des données en mémoire.

Nous avons aussi étudié l'impact de la taillle de la fenêtre d'instruction dans CarCore sur le pire temps d'exécution évalué. Les débits d'entrée et de sortie de la fenêtre étant voisins, la taille n'influe pas sur le pire temps estimé à partir d'une taille de 16 octets par tâche.

Enfin, nous nous sommes intéressés à l'impact de l'ajout d'étages dans le pipeline de CarCore sur l'augmentation du pire temps d'exécution estimé. Il en est ressorti que l'ajout d'étage devait se faire préférentiellement dans la partie commune du pipeline

afin de limiter l'augmentation des latences lors des branchements et des lectures de données en mémoire.

5.2 Perspectives

A court terme, l'analyse du tampon d'écriture et de la fenêtre d'instructions peuvent augmenter la précision du modèle et donc permettre d'obtenir des estimations plus proches du pire temps d'exécution réel des tâches. D'autre part, à l'issue de notre travail, il nous semble intéressant que les perspectives suivantes soient étudiées.

5.2.1 Modifications de CarCore pour mieux exploiter le parallélisme d'instructions

Tout d'abord, on peut étudier des modifications du pipeline de CarCore lui-même, comme le passage de l'étage de décodage dans la partie commune du pipeline, afin de réduire les latences liées aux branchements et aux lectures en mémoire. Au regard des mesures de la section 4.2.2, l'ajout d'un étage avant l'ordonnancement entrainant une augmentation moyenne de 5% du pire temps d'exécution évalué, contre 20% environ pour un ajout après l'ordonnancement, on peut penser que le pire temps d'exécution évalué baisserait d'environ 15% pour les programmes de test que nous avons utilisés.

D'autre part, les mécanismes comme des mémoires caches ou l'exécution spéculative, décrits dans la section 2.2.1 semblent utilisables tant que les structures de stockage sont partagées statiquement. Dans le cas d'un partage dynamique de ces structures entre les tâches, il convient de mener une étude plus approfondie. De même pour l'exécution non ordonnée des tâches, car des contentions pourraient alors se produire entre la tâche de plus haute priorité et les autres. L'isolement temporel de la tâche critique ne serait alors peut-être pas assuré.

5.2.2 Vers une meilleure exploitation du parallélisme de tâche

Pour permettre à CarCore d'exécuter plusieurs tâches temps réel, on pourrait utiliser d'autres algorithmes d'ordonnancement. L'utilisation de la politique du tourniquet strict peut être modélisée avec précision. Elle doit être employée pour les instructions et les microinstructions. Il faut aussi être prudent pour ce qui concerne les opérations qui ont une longue latence comme les lectures en mémoire. Ce type d'opérations, lancées

à la suite par plusieurs tâches pourraient aboutir à un délai supplémentaire difficile à quantifier.

D'autre part, avec cette politique, lorsqu'une tâche est bloquée, son tour est perdu et des bulles sont insérées dans le pipeline. On peut limiter cette perte de performeance en utilisant un ordonnancement par tourniquet optimisé qui ne perd pas ces tours. Mais la modélisation devient alors difficile car on ne peut pas savoir à quels moments les tâches ne passeront pas à leur tour sans connaitre précisément l'entrelacement des tâches. La précision de l'évaluation du pire temps d'exécution s'en trouve diminuée d'autant.

L'utilisation d'autres politiques doit aussi être considérée. On peut penser à des politiques déja utilisées dans les ordonnanceurs logiciels temps réels, mais ceci pose plusieurs problèmes. D'abord beaucoup d'entre eux, comme EDF ou LLF, ont besoin de connaître le pire temps d'exécution pour fonctionner. D'autre part, ce sont des algorithmes qui sont utilisés par des ordonnanceurs de systèmes d'exploitation qui demandent souvent des calculs dont le temps n'est pas compatible avec le temps de cycle des processeurs. Il serait souhaitable d'en concevoir de nouvelles, compatibles avec les contraintes fortes de l'ordonnancement intra-processeur.

Pour contourner certaines difficultés, par exemple avec la politique d'ordonnancement par tourniquet optimisé, l'examen de l'ensemble des tâches co-exécutées pourrait être exploré. Cependant, la multiplication du nombre de tâches et la considération de tous les ordonnancements possibles semblent devoir conduire à une explosion du domaine à explorer et l'évaluation du pire temps d'exécution pourrait demander rapidement trop de mémoire et un temps de calcul rédhibitoire.

Bibliographie

[1] absint - http ://www.absint.com.

[2] Hassan A. Aljifri, Alexander P. Pons, and Moiez A. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *proceedings of IEEE international, performance, computing and communication conference (ICCC'00)*, pages 430–436, february 2000.

[3] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the 8th Euromicro Workshop on Real-time Systems*, pages 102–107, 1996.

[4] Alexis Arnaud and Isabelle Puaut. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th International Conference on Real-Time and Network Systems (RNTS)*, Poitiers, France, May 2006.

[5] Jonathan Barre. *Architectures Multi-Flots Simultanés pour le temps réel strict*. PhD thesis, Pôle de Recherche et d'Enseignement Supérieur de Toulouse, 2008.

[6] Jonathan Barre, Cédric Landet, Christine Rochange, and Pascal Sainrat. Modeling Instruction-Level Parallelism for WCET Evaluation. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Sidney, 16/08/2006-18/08/2006*, pages 61–67, août 2006.

[7] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, RobertStets, and Ben Verghese. Piranha : a scalable architecture based on single-chip multiprocessing. *SIGARCH Comput. Archit. News*, 28(2) :282–293, 2000.

[8] Iain Bate, Guillem Bernat, G. Murphy, and Peter Puschner. Low-level analysis of a portable WCET analysis framework. In *6th IEEE Real-Time Computing Systems and Applications (RTCSA2000)*, pages 39–48, Dec 2000.

[9] Iain Bate and Ralf Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *ECRTS '04 : Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 215–222, 2004.

[10] Guillem Bernat and Alan Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*, 2000.

[11] Guillem Bernat, Antoine Colin, and Stefan Petters. pWCET : a tool for probabilistic worst-case execution time analysis of real-time systems. Technical report, University of York, 2003.

[12] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS '02 : Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 279, 2002.

[13] Armelle Bonenfant, Hugues Cassé, and Mikaël Houston. Interchange format in order to compare results of the otawa and rapita tools. janvier 2009.

[14] Armelle Bonenfant, Marianne de Michiel, and Pascal Sainrat. oRange : A Tool For Static Loop Bound Analysis. In *Workshop on Resource Analysis, University of Hertfordshire, Hatfield, UK, 09/09/08*, 2008.

[15] Claire Burguiere. *Modélisation de la prédiction de branchement pour le calcul de temps d'exécution pire-cas*. PhD thesis, Université Paul Sabatier, juin 2008.

[16] Claire Burguiere and Christine Rochange. Analysing branch mispredictions related to algorithmic constructs. Rapport de recherche 2005-12, IRIT, Université Paul Sabatier, Toulouse, avril 2005.

[17] Claire Burguiere and Christine Rochange. A contribution to branch prediction modeling in WCET analysis. In *DATE '05 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 612–617, 2005.

[18] Claire Burguiere and Christine Rochange. Modélisation d'un prédicteur de branchement bimodal dans le calcul du WCET par la méthode IPET. In *RTS'05 - 13th International Conference on Real-Time Systems , Paris, 05/04/05-06/04/05*, pages 192–210, avril 2005.

[19] Claire Burguiere and Christine Rochange. On the Complexity of Modelling Dynamic Branch Predictors when Computing Worst-Case Execution Times. In *ERCIM/DECOS Workshop on Dependable Embedded Systems, Lübeck, Germany, 28/08/07*, page (on line), septembre 2007.

[20] Hugues Cassé and Christine Rochange. OTAWA, Open Tool for Adaptative WCET Analysis. In *Design, Automation and Test in Europe (Poster session "University Booth") (DATE), Nice, 17/04/07-19/04/07*, avril 2007.

[21] Francisco Cazorla, Peter Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramirez, and Mateo Valero. Architectural support for real-time task scheduling in SMT processors. In *CASES '05 : Proceedings of the 2005 international*

conference on Compilers, architectures and synthesis for embedded systems, pages 166–176, 2005.

[22] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Predictable performance in SMT processors. In *CF '04 : Proceedings of the 1st conference on Computing frontiers*, pages 433–443. ACM Press, 2004.

[23] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernández. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 171–182, 2004.

[24] Francisco J. Cazorla, Alex Ramirez, Mateo Valero, Peter M. W. Knijnenburg, Rizos Sakellariou, and Enrique Fernández. Qos for high-performance SMT processors in embedded systems. *IEEE Micro*, 24(4) :24–31, 2004.

[25] Roderick Chapman, Alan Burns, and Andy Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Syst.*, 11(2) :145–171, 1996.

[26] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[27] Antoine Colin and Guillem Bernat. Scope-tree : A program representation for symbolic worst-case execution time analysis. In *ECRTS '02 : Proceedings of the 14th Euromicro Conference on Real-Time Systems*, page 50, 2002.

[28] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.*, 18(2/3) :249–274, 2000.

[29] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, 2001.

[30] Antoine Colin, Isabelle Puaut, Christine Rochange, and Pascal Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. In *Technique et Science Informatique*, volume 22, pages 651–677. Lavoisier, 2003.

[31] Patrick Crowley and Jean loup Baer. Worst-case execution time estimation of hardware-assisted multithreaded processors. In *Proc. of the 2nd Workshop on Network Processors*, pages 36–47, 2003.

[32] Keith Diefendorff. Compaq Chooses SMT for Alpha. *Microprocessor Report*, 13(16), décembre 1999.

[33] Gautham K. Dorai, Donald Yeung, and Seungryul Choi. Optimizing SMT processors for high single-thread performance. In *Journal of Instruction-Level Parallelism Vol5*, pages 1–35, 2003.

[34] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading : A platform for next-generation processors. *IEEE Micro*, 17(5) :12–19, septembre/octobre 1997.

[35] Magnus Ekman and Per Stenström. Performance and power impact of issue-width in chip-multiprocessor cores. In *ICPP*, pages 359–368, 2003.

[36] Jacob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[37] Jakob Engblom and Andreas Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 146–153, 1998.

[38] Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. Clustered worst-case execution-time calculation. *IEEE Trans. Comput.*, 54(9) :1104–1122, 2005.

[39] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics — A Rigorous and Practical Approach*. 1996 – ISBN : 0534954251.

[40] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT '01 : Proceedings of the First International Workshop on Embedded Software*, pages 469–485, 2001.

[41] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *Workshop on Compilers and Tools for Embedded Systems*, pages 37–46, 1997.

[42] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2-3) :163–189, 1999.

[43] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES '98 : Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 16–30, 1998.

[44] Hans-Gerhard Groß. *Measuring Evolutionary Testability of Real-Time Software*. PhD thesis, Pontypridd, University of Glamorgan, 2000.

[45] Hans-Gerhard Groß. A prediction system for evolutionary testability applied to dynamic execution time analysis. *Information & Software Technology*, 43(14) :855–862, 2001.

[46] Hans-Gerhard Groß, Bryan F. Jones, and David E. Eyres. Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems. *IEEE Proceedings - Software*, 147(2) :25–30, 2000.

[47] Jan Gustafsson and Andreas Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2) :61–74, 1998.

[48] Jan Gustafsson and Andreas Ermedahl. Experiences from applying WCET analysis in industrial settings. In *ISORC '07 : Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 382–392, 2007.

[49] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Algorithms for infeasible path calculation. In *Sixth International Workshop on Worst-Case Execution Time Analysis, (WCET'2006)*, July 2006.

[50] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench : A free, commercially representative embedded benchmark suite. In *WWC '01 : Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, 2001.

[51] Tom R. Halfhill. Intel's Tiny Atom. *Microprocessor Report*, avril 2008.

[52] Christopher Healy, Viresh Rustagi, David Whalley, and Robert Van Engelen. Engelen. supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18 :121–148, 2000.

[53] Christopher Healy and David Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *RTAS '99 : Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium*, page 79, 1999.

[54] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48 :53–70, 1999.

[55] John L. Hennessy and David A. Patterson. *Architecture des ordinateurs : Une Approche quantitative - seconde édition*. Vuibert Informatique, 2001.

[56] John L. Hennessy and David A. Patterson. *Architecture des ordinateurs : Une Approche quantitative - troisième édition*. Vuibert Informatique, 2003.

[57] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach - fourth edition*. Morgan Kaufmann, 2006.

[58] John L. Hennessy and David A. Patterson. *Computers Organisation and Design - third edition*. Morgan Kaufmann, 2007.

[59] Infineon. *TriCore 1 32-Bit Unified Processor Cores, volume 2, V1.3 Instruction Set*, october 2005.

[60] Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. Ibm power5 chip : A dual-core multithreaded processor. *IEEE Micro*, 24(2) :40–47, 2004.

[61] Shinpei Kato, Hidenori Kobayashi, and Nobuyuki Yamasaki. U-link scheduling : Bounding execution time of real-time tasks with multi-case execution time on SMT processors. In *RTCSA '05 : Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 193–197, 2005.

[62] Kevin B. Kenny and Kwei-Jay Lin. Measuring and analyzing real-time performance. *IEEE Software*, 8(5) :41–49, 1991.

[63] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *RTAS '96 : Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 230–240, 1996.

[64] Raimund Kirner and Peter Puschner. Transformation of path information for WCET analysis during compilation. In *ECRTS '01 : Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 29, 2001.

[65] Raimund Kirner, Peter Puschner, and Ingomar Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. 4th Euromicro International Workshop on WCET Analysis*, pages 67–70, June 2004.

[66] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using Measurements as a Complement to Static Worst-Case Execution Time Analysis. In *Intelligent Systems at the Service of Mankind*, volume 2. Dec. 2005.

[67] Lo Ko, David B. Whalley, and Marion G. Harmon. Supporting user-friendly analysis of timing constraints. *SIGPLAN Not.*, 30(11) :99–107, 1995.

[68] Apostolos A. Kountouris. Safe and efficient elimination of infeasible execution paths in WCET estimation. In *Proceedings of RTCSA'96. IEEE, IEEE Computer*, 1996.

[69] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip multiprocessing. In *MICRO 37 : Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 195–206, 2004.

[70] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures : Understanding mechanisms, overheads and scaling. *SIGARCH Comput. Archit. News*, 33(2) :408–419, 2005.

[71] Hung Q. Le, Wiliam J. Starke, J. Stephen Fields, Francis P. O'Connell, Dung Q. Nguyen, Bruce J. Ronchetti, Wolfram M. Sauer, Eric M. Schwarz, and Michael T. Vaden. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, 51(6) :639–662, 2007.

[72] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Syst.*, 29(1) :27–58, 2005.

[73] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3) :195–227, 2006.

[74] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95 : Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, 1995.

[75] Yanhong A. Liu and Gustavo Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems, volume 1474 of Lecture Notes in Computer Science*, pages 31–40, 1998.

[76] Sonia Lopez, Steve Dropsho, David H. Albonesi, Oscar Garnica, and Juan Lanchares. Rate-driven control of resizable caches for highly threaded SMT processors. In *PACT '07 : Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 416, 2007.

[77] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Processors with Cache Memories*. PhD thesis, Chalmers Univerity of Technology, June 2002.

[78] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Syst.*, 17(2-3) :183–207, 1999.

[79] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. *Real-Time Systems Symposium, IEEE International*, 0 :12, 1999.

[80] Dragan Macos and Frank Mueller. Integrating gnat/gcc into a timing analysis environment. In *10th EUROMICRO Workshop on Real Time Systems*, pages 15–18, 1998.

[81] Levy Markus. Keynote talk #1- eembc and the purposes of embedded processor benchmarking. In *ISPASS '05 : Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, page 1, 2005.

[82] Deborah. T. Marr, Frank Binns, David L. Hill, Glann Hinton, David A. Koufati, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 06(01) :4–15, 2002.

[83] Harry M. Mathis, Alex E. Mericas, John D. McCalpin, Richard J. Eickemeyer, and Steven R. Kunkel. Characterization of simultaneous multithreading (SMT) efficiency in power5. *IBM Journal of Research and Development*, 49(4/5) :555–564, 2005.

[84] Phil Mcminn. Search-based software test data generation : A survey. *Software Testing, Verification and Reliability*, 14(2) :105–156, 2004.

[85] Aloysius Mok, Prasanna Amerasinghe, Moyer Chen, and Kamtorn Tantisirivat. Evaluating tight execution time bounds of programs by annotations. *IEEE Real-Time System Newsletter*, 5(2-3) :81–86, 1989.

[86] Frank Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2-3) :217–247, 2000.

[87] Frank Mueller and Joachim Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Real-Time Systems*, pages 144–154. IEEE, 1998.

[88] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. Papabench : a free real-time benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[89] University of Mälardalen. Worst case execution time project / benchmarks – http ://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[90] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII : Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, 1996.

[91] Greger Ottosson and Mikael Sjodin. Worst case execution time analysis for modern hardware architectures. In *Proc. of ACM SIGPLAN, Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 47–55, 1997.

[92] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA '09 : Proceedings of the 36th annual international symposium on Computer architecture*, pages 57–68, 2009.

[93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5(1) :31–62, 1993.

[94] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5) :48–57, 1991.

[95] Shang Yun Park. *Predicting deterministic execution tims for real-time programs*. PhD thesis, University of Washington, Seattle, WA, 1992.

[96] Kaustubh Patil, Kiran Seth, and Frank Mueller. Compositional static instruction cache simulation. In *LCTES '04 : Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 136–145, 2004.

[97] Stefan M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technische Universität München, München, 2002.

[98] Preston, Ronald P. and Badeau, Roy W. and Bailey,Daniel W. and Bell, Shane L. and Biro, Larry L. and Bowhill, William J. and Dever, Daniel E. and Felix, Stephen and Gammack, Richard and Germini, Valeria and Gowan, Michael K. and Gronowski, Paul and Jackson, Daniel B. and Mehta, Shekhar and Morton, Shannon V. and Pickholtz, Jeffrey D. and Reilly, Matthew H. and Smith, Michael J. Design of an 8-wide superscalar risc microprocessor with simultaneous multithreading. In *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International*, pages 266–500, 2002.

[99] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02 : Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 114–123, 2002.

[100] Peter Puschner. A tool for high-level language analysis of worst-case execution times. In *10th Euromicro Workshop on Real-Time Systems*, pages 130–137, June 1998.

[101] Peter Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2) :159–176, September 1989.

[102] Peter Puschner and Anton Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universitat Wien, 1995.

[103] Steven E. Raasch and Steven K. Reinhardt. The impact of resource partitioning on SMT processors. In *PACT '03 : Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 15, 2003.

[104] Heckmann Reinhold, Langenbach Marc, Thesing Stephan, and Wilhelm Reinhard. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7) :1038–1054, july 2003.

[105] Christine Rochange and Pascal Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Transactions on High-Performance Embedded Architecture and Compilation*, 2(3) :109–128, 2007.

[106] Erven Rohou, François Bodin, André Seznec, Gwendal Le Fol, François Charot, and Frédéric Raimbault. Salto : System for assembly-language transformation and optimization. In *6th Workshop on Compilers for Parallel Computers*, pages 261–272, 1996.

[107] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. *Real-Time Systems Symposium, IEEE International*, 0 :49–60, 2007.

[108] Alex Settle, Joshua Kihm, Andrew Janiszewski, and Dan Connors. Architectural support for enhanced SMT job scheduling. In *PACT '04 : Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–73, 2004.

[109] Kato Shimpei, Kobayashi Hidenori, and Yamasaki Nobuyuki. Real-time scheduling for SMT processors. *Joho Shori Gakkai Shinpojiumu Ronbunshu*, 2005(18) :109–118, 2005.

[110] Ronak Singhal. Inside intel next generation nehalem microarchitecture. Technical report, Inter Developer Forum, 2008.

[111] Balaram Sinharoy, Ronald N. Kalla, Joel M. Tendler, Richard J. Eickemeyer, and Jody B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5) :505–521, 2005.

[112] William Stallings. *Computer Organization and Architecture : Designing for Performance - seventh edition*. Prentice Hall, 2005.

[113] John A. Stankovic. Misconceptions about real-time computing : a serious problem for next-generation systems. *IEEE Computer*, 21(10) :10–19, 1988.

[114] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4) :339–355, 2000.

[115] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES '01 : Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 132–140, 2001.

[116] Henrik Theiling, Christian Ferdinand, Absint Angewandte Informatik Gmbh Saarbrucken, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18 :157–179, 2000.

[117] Nigel J. Tracey, John A. Clark, and Keith C. Mander. The Way Forward for Unifying Dynamic Test Case Generation : The Optimisation-based Approach. In *IFIP International Workshop on Dependable Computing and its Applications (DCIA 98)*, Janvier 1998.

[118] Yau tsun Steven, Li Sharad, and Malik Andrew Wolfe. Cache modeling for real-time software : beyond direct mapped instruction caches. In *RTSS '96 : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, pages 254–263, 1996.

[119] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice : Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA '96 : Proceedings of the 25th annual international symposium on Computer architecture*, pages 191–202, 1996.

[120] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading : maximizing on-chip parallelism. In *ISCA '95 : Proceedings of the 22nd annual international symposium on Computer architecture*, 1995.

[121] Sascha Uhrig, S. Maier, and Theo Ungerer. Toward a processor core for real-time capable autonomic systems. *International Symposium on Signal Processing and Information Technology*, 0 :19–22, 2005.

[122] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *SIGMETRICS '03 : Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, 2003.

[123] Alexander Vrchoticky. *The basis for static execution time prediction*. PhD thesis, Vienna University of Technology, 1994.

[124] Alexander Vrchoticky. Compilation support for fine-grained execution time analysis. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1994.

[125] Nicky Wiliams, Bruno Marre, and Patricia Mouy. Interleaving static and dynamic analises to generate path tests for c functions. In *5th Workshop on System Testing and Validation (SV'04)*, 2004.

[126] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *5th European Dependable Computing Conference*, pages 281–292, 2005.

[127] Fabian Wolf, Jan Staschulat, and Rolf Ernst. Associative caches in formal software timing analysis. In *DAC '02 : Proceedings of the 39th conference on Design automation*, pages 622–627, 2002.

Annexe A

Preuves des équations

A.1 preuve de l'équation 3.2

soient $\beta \in R$ et et $\pi \in R$, les ressources les plus critiques pour les nœuds L_B et L_P respectivement :

$$\begin{cases} \rho_{L_B} = d^{\beta}_{LB} + a^{\beta} \\ \rho_{L_P} = d^{\pi}_{LP} + a^{\pi} \end{cases}$$

et

$$\Delta(L_B, L_P) = d^{\beta}_{LB} + a^{\beta} - d^{\pi}_{LP} - a^{\pi}$$

Si le nœud L_P dépend aussi de la ressource β, ($e^{\beta}_{L_P} = 1$), on peut écrire :

$$d^{\beta}_{LB} + a^{\beta} \leq d^{\pi}_{LP} + a^{\pi}$$

Comme β n'est pas la ressource la plus critique pour L_P (ou $\beta = \pi$). alors :

$$a^{\beta} \leq d^{\pi}_{LP} + a^{\pi} - d^{\beta}_{LP}$$

et

$$\Delta(L_B, L_P) \leq d^{\beta}_{LB} - d^{\beta}_{LP} \tag{A.1}$$

sinon, ($e^{\beta}_{L_P} = 0$), on remarque que $\forall \beta \in R, a^{\beta} \leq a^{\lambda} - \alpha_{\beta}$, donc :

$$\Delta(L_B, L_P) \leq d^{\beta}_{LB} + a^{\lambda} - \alpha_{\beta} - d^{\pi}_{LP} - a^{\pi}$$

De plus, λ n'est pas la ressource la plus critique pourle nœud L_P (ou $\lambda = \pi$). Alors, $d^{\lambda}_{L_P} + a^{\lambda} \leq d^{\pi}_{L_P} + a^{\pi}$ et $a^{\lambda} \leq d^{\pi}_{L_P} + a^{\pi} - d^{\lambda}_{L_P}$. Donc :

$$\Delta(L_B, L_P) \leq d^{\beta}_{LB} - d^{\lambda}_{L_P} - \alpha_{\beta} \tag{A.2}$$

Finalement, les équations A.1 et A.2 peuvent être rassemblées en :

$$\Delta(L_B, L_P) \leq max_{r \in R}(e^{r}_{L_B}.(d^{r}_{L_B} - e^{r}_{L_P}.d^{r}_{L_P} - (1 - e^{r}_{L_P}).(d^{\lambda}_{L_P} + \alpha_r)))$$

Table des figures

2.1 Le temps d'exécution d'une tâche 9
2.2 Code source d'un programme analysé 12
2.3 Représentation du flot de contrôle d'un programme 13
2.4 Exemple de système de contraintes généré par la méthode d'énumération des chemins implicites 17
2.5 Préfixe et suffixe d'un bloc de base 19
2.6 Le recouvrement des blocs de base 20
2.7 Processeur moderne 22
2.8 Les avantages des processeurs multiflots 32
2.9 Partage des structures de stockage 34
2.10 Architecture type d'un processeur bi-cœur homogène 40

3.1 Le pipeline de CarCore 46
3.2 Le chargement des instructions 47
3.3 Un exemple de processeur et de code pour illustrer la construction des graphes d'exécution 50
3.4 exemple de graphe d'exécution 51
3.5 Modélisation du chargement des instructions de taille variable 54
3.6 Une instruction pour le pipeline adresse suivie de n'importe quelle autre 54
3.7 Une instruction pour le pipeline entier suivie d'une instruction pour le pipeline adresse 54
3.8 vue d'ensemble de la production d'un emulateur par GLISS 58
3.9 exemple de déclaration des ressources 59
3.10 exemple de déclaration des mode d'adressage 60
3.11 Exemple de déclaration d'une instruction 61
3.12 exemple d'utilisation du pré-décodage 61
3.13 exemple d'utilisation de la fonction "coerce" 62
3.14 L'empilement de couches du système 63

3.15 La couche d'abstraction de l'architecture 64
3.16 Description du processeur CarCore en XML 67

4.1 Fichier d'informations de flots produit par mkff pour le programme bs . 75
4.2 pire temps d'exécution évalué en fonction de la taille de la fenêtre d'instructions 81
4.3 Augmentation du pire temps d'exécution estimé pour la configuration P-1 84
4.4 Augmentation du pire temps d'exécution estimé pour la configuration P-2 84
4.5 Augmentation du pire temps d'exécution estimé pour la configuration P-3 85
4.6 Augmentation du pire temps d'exécution estimé pour la configuration P-4 85
4.7 Pour les configurations P-1 et P-2, les graphes d'exécution sont les mêmes 86
4.8 Augmentation moyenne pour chaque configuration 87
4.9 Augmentation moyenne pour chaque programme de test 88
4.10 Impact de l'ajout d'étages, configuration P-3 89
4.11 Impact de l'ajout d'étages, configuration P-4 89

Liste des tableaux

3.1 Résumé de l'interface de l'émulateur généré par GLISS 59
3.2 Les instructions codées dans le chargeur pour TriCore/CarCore 66

4.1 Quelques caractéristiques des benchmarks de l'université de Mälardalen 73
4.2 Les attributs possibles dans le format f4 74
4.3 Caractéristiques statiques des programmes de test 76
4.4 Caractéristiques dynamiques des programmes de test 76
4.5 surestimation du pire temps d'exécution estimé par rapport au temps mesuré 78
4.6 Statistiques de surestimation des blocs de base des programmes de test . 79
4.7 Les configurations de tests pour la taille de la fenêtre d'instructions . . . 81
4.8 Répartition de l'augmentation du pire temps d'exécution des blocs de base dans la configuration F-8 82